INSTITUTO DE ENGENHARIA NUCLEAR

FABIO WAINTRAUB

MODELO COMPUTACIONAL PARALELO EM GPU PARA CORREÇÃO DE PLUMAS RADIOATIVAS UTILIZANDO MEDIDAS DE CAMPO

Rio de Janeiro

2019

FABIO WAINTRAUB

MODELO COMPUTACIONAL PARALELO EM GPU PARA CORREÇÃO DE PLUMAS RADIOATIVAS UTILIZANDO MEDIDAS DE CAMPO

Dissertação apresentada ao Programa de Pós-graduação em Ciência e Tecnologia Nucleares do Instituto de Engenharia Nuclear da Comissão Nacional de Energia Nuclear como parte dos requisitos necessários para a obtenção do Grau de Mestre em Ciência em Engenharia Nuclear – Profissional em Métodos Computacionais

Orientadores: Prof. Dr. Claudio Marcio do Nascimento Abreu Pereira

Prof. Dr. Carlos Alexandre Fructuoso Jorge

Prof. Dr. Adino Americo Heimlich Almeida

Rio de Janeiro

2019

WAINTRAUB, Fabio

Modelo computacional paralelo em GPU para correção de plumas radioativas utilizando medidas de campo / Fabio Waintraub – Rio de Janeiro: CNEN/IEN, 2019.

xiii, 65f. : il.; 31 cm.

Orientadores: Claudio Marcio do Nascimento Abreu Pereira e Carlos Alexandre Fructuoso Jorge e Adino Americo Heimlich Almeida

Dissertação (Mestrado em Ciência e Tecnologia Nucleares) – Instituto de Engenharia Nuclear, PPGIEN, 2019.

1.Computação Paralela. 2. GPU. 3. Estimativa de Dose

MODELO COMPUTACIONAL PARALELO EM GPU PARA CORREÇÃO DE PLUMAS RADIOATIVAS UTILIZANDO MEDIDAS DE CAMPO

Fabio Waintraub

DISSERTAÇÃO SUBMETIDA AO PROGRAMA DE PÓS GRADUAÇÃO EM CIÊNCIA E TECNOLOGIA NUCLEARES DO INSTITUTO DE ENGENHARIA NUCLEAR DA COMISSÃO NACIONAL DE ENERGIA NUCLEAR COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA NUCLEAR – PROFISSIONAL EM MÉTODOS COMPUTACIONAIS.

Aprovada por:

_____

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

_____

Prof. Carlos Alexandre Fructuoso Jorge, D.Sc.

_____

Prof. Adino Americo Heimlich Almeida, D. Sc.

_____

Prof. André Przewodowski Filho, D. Sc.

_____

Prof. Antônio Carlos de Abreu Mol, D. Sc.

RIO DE JANEIRO, RJ – BRASIL.

ABRIL DE 2019

AGRADECIMENTOS

Aos amigos, familiares e a todos que ajudaram ou incentivaram de alguma forma a realização desse projeto.

RESUMO

Em caso de acidentes na usina com liberação de material radioativo para o meio externo, o sistema de dispersão atmosférica de radionuclídeos (SDAR) deve ser capaz de estimar as distribuições espaciais de concentrações dos radionuclídeos e de taxas de dose. Porém, em casos de acidentes nucleares severos a usina tende a ser levada à condições extremas. Nesses casos a progressão do acidente pode se tornar imprevisível, o que pode prejudicar de forma significativa o processo de tomada de decisão do operador.

Dentre as investigações mais recentes para contornar esse problema, pode-se destacar o trabalho científico onde foi proposta a utilização de uma matriz de correção a ser aplicada aos mapas de distribuição de doses e taxas de doses estimadas originalmente, de forma a gerar uma distribuição corrigida que melhor represente as medidas de campo. O método proposto apresentou resultados motivadores, entretanto, devido à complexidade do problema, foram utilizadas ferramentas de elevado custo computacional o que tornou a sua aplicação restrita em casos práticos. Para contornar esse problema e viabilizar o uso de resoluções maiores do mapa de doses, propõe-se neste trabalho, a utilização de um modelo paralelo de processamento, utilizando uma Unidade de Processamento Gráfico (GPU).

Como resultado, para resolução inicial de (67x43) o método GPU apresenta ganho de 19,66 vezes em relação ao tempo de execução do sistema em CPU e esse ganho só cresce quando aumentamos a resolução do mapa usado pelo sistema, o que nos permite não só aumentar a resolução usada, mas também o numero de execuções possíveis de serem feitas pelo operador do sistema dentro da janela temporal definida de 15 minutos, permitindo a aplicação prática do método ao problema real, conforme o objetivo inicial dessa dissertação.

Palavras-Chave: Estimativa de Dose, GPU, Computação Paralela**.**

ABSTRACT

In case of accidents at a nuclear power plant with release of radioactive material to the external environment, the radionuclide dispersion system (SDAR) should be able to estimate, the spatial distributions of radionuclide concentrations and dose rates. However, in cases of severe nuclear accidents the plant tends to be brought to extreme conditions. In such cases the progression of the accident may become unpredictable, which can significantly impair the operator's decision-making process.

Among the most recent investigations to circumvent this problem, one can highlight the cientific work where it was proposed the use of a correction matrix to be applied directly to the maps of dose distribution and dose rates estimated originally, in order to generate a corrected distribution that best represents the measured field measurements. The proposed method presented motivating results, however, due to the complexity of the problem, tools of high computational cost were used, which made its application restricted in practical cases. In order to overcome this problem and to make possible the use of higher resolutions to describe the dose map, we propose the use of a parallel processing model using a GPU.

As a result, for the initial resolution of (67x43) the GPU method presents a gain of 19.66 times in relation to the execution time of the CPU system and this gain only increases when we increase the resolution of the map used by the system, which allows us only increase the resolution used, but also the number of executions possible to be made by the system operator within the defined time window of 15 minutes, allowing the practical application of the method to the real world problem, fulfilling the initial objective of this dissertation.

Keywords: Dose Estimation, GPU, Parallel Computing.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

CNAAA	- Central Nuclear Almirante Álvaro Alberto
CPU	- Central Processing Unit (Unidade Central de Processamento)
CUDA	- Compute Unified Device Architecture
DAR	- Dispersão Atmosférica de Radionuclídeos
GPU	- Graphics Processor Unit (Unidade Gráfica de Processamento)
SDAR	- Sistema de Dispersão Atmosférica de Radionuclídeos
ULA	- Unidade Lógica e Aritmética
WNA	- World Nuclear Association

SUMÁRIO

1. INTRODUÇÃO

1.1. APRESENTAÇÃO DO PROBLEMA E CONTEXTUALIZAÇÃO

A energia nuclear é uma das principais fontes geradoras de energia elétrica atuais. Segundo a (INTERNATIONAL ENERGY AGENCY, 2017), ela é responsável por 10,6% da produção de energia elétrica no mundo.

A segurança nesse setor é um aspecto prioritário e como tal, em princípio não deveria ser comprometida por qualquer motivo. Desse modo, para que uma usina nuclear opere é necessário que esta atenda ao cumprimento e manutenção dos diversos critérios de segurança, definidos por órgãos governamentais e por organismos internacionais. Entretanto, ainda assim, as usinas encontram-se susceptíveis a eventuais falhas. Segundo a *World Nuclear Association* (WNA) entre os anos 1952-2017 ocorreram 11 acidentes graves envolvendo reatores nucleares. A exemplo disso, pode-se mencionar o acidente em Chernobyl (Ucrânia – 1986): tido como o maior acidente nuclear da história. O acidente teve como causa um conjunto de fatores (design falho de reator, erros de operação e falha humana). Durante o acidente, o reator 4 explodiu, causando a liberação de uma grande quantidade de material radioativo para atmosfera, o que formou uma pluma radioativa que percorreu e afetou uma parte significativa da Europa.

Em caso de acidentes na usina com liberação de material radioativo para o meio externo, como o citado acima, são utilizados modelos de dispersão atmosférica para prever o transporte e difusão dos radionuclídeos dispersos para a atmosfera, de forma a calcular os impactos desta liberação para o meio ambiente e para o direcionamento da evacuação da população próxima à central nuclear ou no caminho da pluma radioativa.

Um Sistema de Dispersão Atmosférica de Radionuclídeos (SDAR) pode ser dividido basicamente em 4 módulos básicos, como é o caso do sistema utilizado nas usinas nucleares brasileiras, ilustrado na Figura 1:

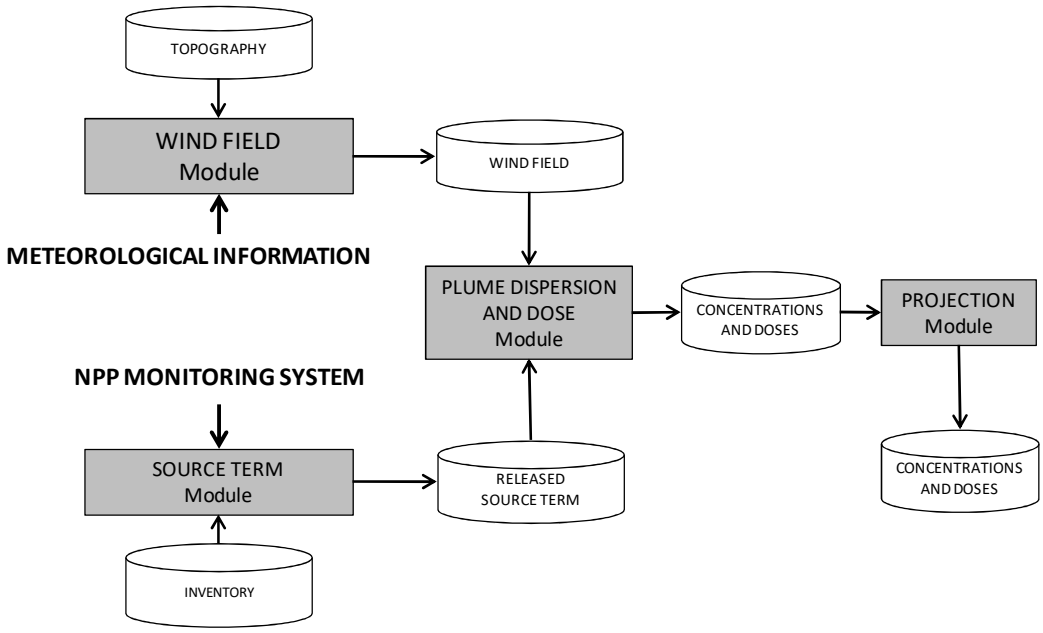

Figura 1 – Diagrama esquemático do SDAR utilizado nas usinas brasileiras, (Carvalho dos Santos, M. et al ,2018).

Source Term Module: estima a quantidade e a taxa de material nuclear liberado (*Source Term*) com base no inventário atual e nos status da central nuclear. *Wind Field Module*: utiliza informações topográficas e meteorológicas para produzir o campo de vento, usando um campo de velocidade tridimensional não divergente. *Plume Dispersion and Dose Module*: usa as saídas dos módulos Termo Fonte e do Campo de Vento para simular a dispersão de plumas e calcular as doses equivalentes, baseado em (FABRICK. SKLAREW e WILSON, 1987). E finalmente, o *Projection Module*: faz projeções da dispersão da pluma em etapas de tempo futuro.

O SDAR deve ser capaz de estimar, com a melhor precisão possível, as distribuições espaciais de concentrações dos radionuclídeos e de taxas de dose, de forma a prover apoio adequado às equipes de emergência. Porém, em casos de acidentes nucleares severos a usina tende a ser levada a condições extremas, além das originalmente projetadas. Nesses casos, segundo (McKenna & Giitter, 1988), a progressão do acidente pode se tornar imprevisível. Logo, qualquer tentativa de se estimar o termo fonte (caracterização da liberação de material radioativo) pode ser imprecisa em varias escalas de grandeza, o que pode levar a uma estimativa da distribuição espacial de doses bastante distorcida da realidade, prejudicando de forma significativa o processo de tomada de decisão do operador.

A fim de melhorar as estimativas de dose ao público nos casos onde a previsão através de um SDAR é errônea, diversas abordagens têm sido investigadas baseadas em medidas de campo para melhor estimar ou corrigir a pluma gerada por um possível acidente nuclear.

Em (Athey, Brandon, & Jr., 2013): aplicou-se um fator de proporcionalidade, razão entre valor medido e valor inicialmente estimado, ao termo fonte.

(Chow et al, 2008) estimaram a dispersão da pluma em ambientes urbanos usando medidas de concentração obtidas na direção do vento e modelos de Simulação conhecidos.

No trabalho de (Zheng e Chen, 2011) utilizou-se um método de busca de padrões baseado em medições obtidas na direção do vento, a fim de prever características do termo fonte de liberação de materiais perigosos e suas concentrações.

Dentre as investigações mais recentes, pode-se destacar o trabalho de (Przewodowski Filho, A. et al 2017) onde foi proposta a utilização de uma matriz de correção (baseada na

concatenação de transformações geométricas, ponderadas por um fator de multiplicação) a ser aplicada aos mapas de distribuição de doses e taxas de doses estimados originalmente e possivelmente errados, de forma a gerar uma distribuição corrigida que melhor represente as medidas de campo. O método proposto apresentou resultados motivadores, entretanto, devido à complexidade do problema, foram utilizadas ferramentas de elevado custo computacional, o que tornou a sua aplicação restrita em casos práticos.

1.2. OBJETIVO

Embora o modelo atual (Przewodowski Filho, A. et al 2017) tenha se mostrado capaz de resolver o problema simplificado. Porém se vislumbramos uma descrição mais realista na resolução do mapa de distribuição de dose, esse modelo se mostra muito custoso em termos de processamento como podemos ver na Tabela 1, que exibe a resolução do domínio computacional (Número de células na direção X versus Número de células na direção Y) e o respectivo tempo de execução na CPU, mesma máquina usada no restante do trabalho.

Tabela 1 - Tempos de execução CPU.

Resolução do mapa	**Tempo Execução**
(67 x 43)	00:07 min
(335 x 215)	03:02 mim
(670 x 430)	19:50 mim
(1005 x 645)	50:19 min

Levando em conta os tempos de execução medidos e a largura da janela temporal disponível para os cálculos (atualização do SDAR), 15 minutos, o modelo atual (Przewodowski Filho, A. et al 2017) se mostrou suficiente para a primeira resolução (67 x 43), podendo rodar ate 128 avaliações dentro da janela temporal, já para a resolução de (335 x

215), o numero de avaliações possíveis cai para 4 e para resoluções maiores que (670 x 430) o modelo se torna incapaz de ser executado na janela temporal disponível.

Para contornar esse problema e viabilizar o uso de resoluções maiores do mapa de doses, propõe-se a utilização de um modelo paralelo de processamento, utilizando uma Unidade de Processamento Gráfico (GPU) para lidar com os cálculos no lugar da uma Unidade Central de Processamento (CPU).

Para tal, um novo programa, utilizando recursos específicos da GPU, precisou ser desenvolvido com base em uma arquitetura computacional própria para programação de GPU, a Compute Unified Device Architecture (CUDA). A estrutura algorítmica do novo programa precisou ser completamente reformulada, de forma a atender à arquitetura paralela utilizada.

Neste trabalho, além de uma visão macro da arquitetura computacional utilizada, são descritos em detalhes todos os aspectos relacionados com a reformulação dos algoritmos utilizados no programa. A eficiência computacional do novo programa é analisada à luz das necessidades práticas e comparada com a versão não paralela.

2. FUNDAMENTAÇÃO TEÓRICA

Nessa seção será abordado um pouco do histórico e funcionamento de alguns conceitos utilizados no projeto a ser desenvolvido que precisam ser melhor entendidos para compreensão do trabalho como um todo. São eles: transformadas geométricas, filtro por difusão, otimização por enxame de partículas (PSO) e computação paralela em GPU.

2.1. TRANSFORMADAS GEOMÉTRICAS

Segundo o método descrito por (Przewodowski Filho, A. et al 2017), na construção da transformada final foram usadas transformadas de translação, rotação e escala de forma combinada. Além de definirmos um ponto central de interesse, como vemos na Figura 2:

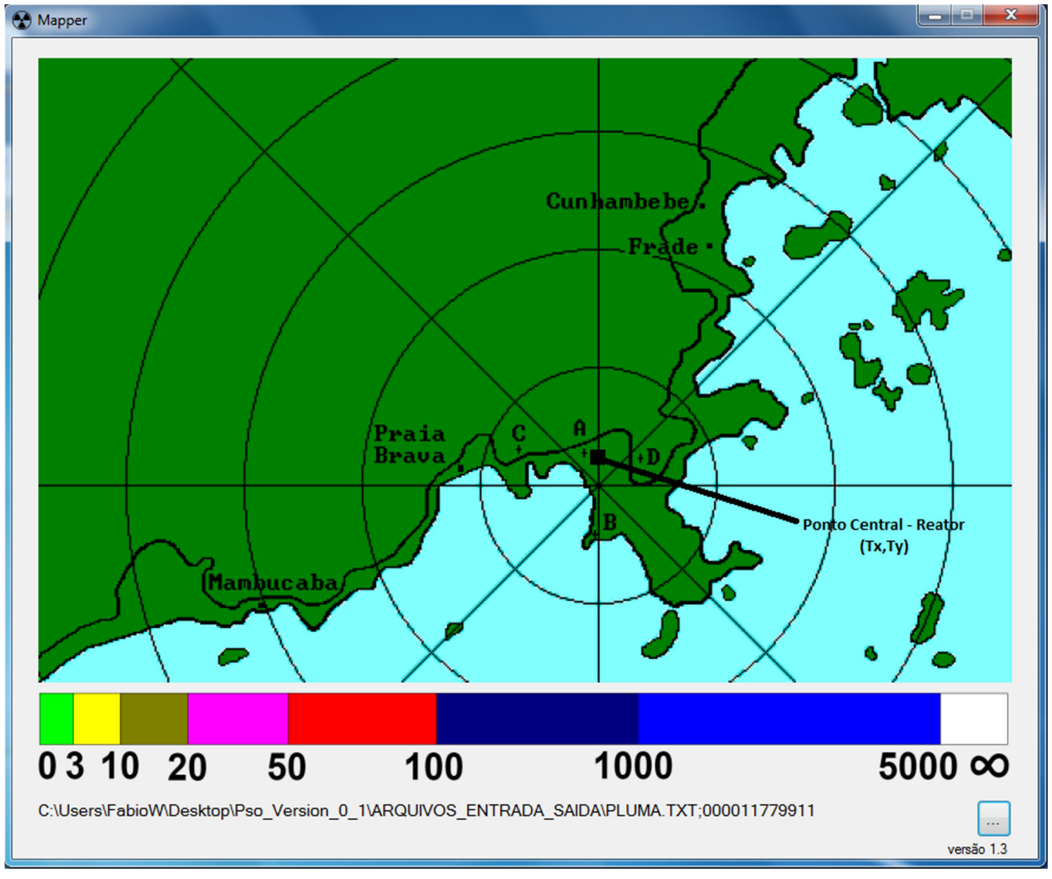

Figura 2 - Ponto de origem (reator)

Primeiramente aplicamos a matriz de rotação em X à matriz de translação para origem, como vemos abaixo:

$$\begin{bmatrix} cos\,\theta & \text{sen}\,\theta & 0 \\ -sen\,\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Tx & Ty & 1 \end{bmatrix} = \begin{bmatrix} cos\,\theta & \text{sen}\,\theta & 0 \\ -sen\,\theta & \cos\theta & 0 \\ Tx & Ty & 1 \end{bmatrix} \quad (1)$$

Depois, aplicamos a matriz resultante da multiplicação anterior à matriz de escala, como vemos abaixo:

$$\begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} cos\,\theta & \text{sen}\,\theta & 0 \\ -sen\,\theta & \cos\theta & 0 \\ Tx & Ty & 1 \end{bmatrix} = \begin{bmatrix} cos\,\theta * \text{Sx} & \text{sen}\,\theta * \text{Sx} & 0 \\ -sen\,\theta * \text{Sy} & \cos\theta * \text{Sy} & 0 \\ Tx & Ty & 1 \end{bmatrix} \quad (2)$$

Em seguida, aplicamos a matriz resultante da multiplicação anterior pela matriz de translação de volta para a posição original da pluma, como vemos abaixo:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -Tx & -Ty & 1 \end{bmatrix} \begin{bmatrix} cos\,\theta * \text{Sx} & \text{sen}\,\theta * \text{Sx} & 0 \\ -sen\,\theta * \text{Sy} & \cos\theta * \text{Sy} & 0 \\ Tx & Ty & 1 \end{bmatrix} =$$

$$\begin{bmatrix} cos\,\theta * \text{Sx} & \text{sen}\,\theta * \text{Sx} & 0 \\ -sen\,\theta * \text{Sy} & \cos\theta * \text{Sy} & 0 \\ -Tx * cos\,\theta * \text{Sx} + \text{Ty} * sen\,\theta * \text{Sy} + \text{Tx} & -Tx * sen\,\theta * \text{Sx} - \text{Ty} * cos\,\theta * \text{Sy} + \text{Ty} & 1 \end{bmatrix} \quad (3)$$

Após isso, aplicamos a matriz resultante da multiplicação anterior pela matriz de translação independente do ponto de origem, como vemos abaixo:

$$\begin{bmatrix} cos\,\theta * Sx & sen\,\theta * Sx & 0 \\ -sen\,\theta * Sy & cos\,\theta * Sy & 0 \\ -Tx * cos\,\theta * Sx + Ty * sen\,\theta * Sy + Tx & -Tx * sen\,\theta * Sx - Ty * cos\,\theta * Sy + Ty & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Tpx & Tpy & 1 \end{bmatrix} =$$

$$\begin{bmatrix} cos\,\theta * Sx & sen\,\theta * Sx & 0 \\ -sen\,\theta * Sy & cos\,\theta * Sy & 0 \\ -Tx * cos\,\theta * Sx + Ty * sen\,\theta * Sy + Tx + Tpx & -Tx * sen\,\theta * Sx - Ty * cos\,\theta * Sy + Ty + Tpy & 1 \end{bmatrix} \quad (4)$$

Por último, aplicamos a matriz resultante da multiplicação anterior à matriz posição simulada das doses para calcular suas posições atuais e adequá-las pela constante *K*, como vemos abaixo:

$$[x\ y\ 1] * \begin{bmatrix} cos\,\theta * \mathrm{Sx} & \mathrm{sen}\,\theta * \mathrm{Sx} & 0 \\ -sen\,\theta * \mathrm{Sy} & \cos\theta * \mathrm{Sy} & 0 \\ -Tx * cos\,\theta * \mathrm{Sx} + \mathrm{Ty} * sen\,\theta * \mathrm{Sy} + \mathrm{Tx} + Tpx & -Tx * sen\,\theta * \mathrm{Sx} - \mathrm{Ty} * cos\,\theta * \mathrm{Sy} + \mathrm{Ty} + Tpy & 1 \end{bmatrix} =$$

$$\begin{bmatrix} x * cos\,\theta * \mathrm{Sx} - \mathrm{y} * sen\,\theta * \mathrm{Sy} - Tx * cos\,\theta * \mathrm{Sx} + \mathrm{Ty} * sen\,\theta * \mathrm{Sy} + \mathrm{Tx} + Tpx \\ x * \mathrm{sen}\,\theta * \mathrm{Sx} + \mathrm{y} * \cos\theta * \mathrm{Sy} - Tx * sen\,\theta * \mathrm{Sx} - \mathrm{Ty} * cos\,\theta * \mathrm{Sy} + \mathrm{Ty} + Tpy \\ 1 \end{bmatrix}' \quad (5)$$

Logo, como a matriz usada para definir o espaço amostral é 2D, temos que:

$$\begin{bmatrix} x * cos\,\theta * Sx - y * sen\,\theta * Sy - Tx * cos\,\theta * Sx + Ty * sen\,\theta * Sy + Tx + Tpx \\ x * sen\,\theta * Sx + y * cos\,\theta * Sy - Tx * sen\,\theta * Sx - Ty * cos\,\theta * Sy + Ty + Tpy \end{bmatrix}' = \begin{bmatrix} x \\ y \end{bmatrix}' * K \quad (6)$$

Após a aplicação de todas as operações descritas acima sobre a matriz original simulada, a matriz gerada que representa a pluma corrigida pode apresentar regiões vazias, as quais devem ser preenchidas para que possamos ter um resultado mais homogêneo.

Portanto o que se pretende é uma transformação linear ou um filtro que aplicado na Matriz Transformada represente a pluma e respeite a natureza dos fenômenos envolvidos em seu comportamento ao longo do tempo, sem efetuar os custosos cálculos por modelos de mecânica dos fluidos.

2.2. FILTRO POR DIFUSÃO

Como vimos na secção anterior, o resultado da aplicação das transformadas geométricas pode possuir descontinuidades. Para preenchê-las, propormos nesse trabalho usar a solução por diferença finita para a Eq. de calor para um domínio 2D com fronteiras conhecidas.

À partir da equação de calor ,temos que:

$$\rho Cp\left(\frac{\partial T}{\partial t}\right) = k\left(\frac{\partial^2 T}{\partial x}^2 + \frac{\partial^2 T}{\partial y}^2\right) \quad (7)$$

$$\left(\frac{\partial T}{\partial t}\right) = \alpha\left(\frac{\partial^2 T}{\partial x}^2 + \frac{\partial^2 T}{\partial y}^2\right) \quad (8)$$

Discretizando a Eq. Acima, obtemos:

$$(T_{i,j}^{K+1} - T_{i,j}^{k})/\Delta t = \alpha\left[\left(T_{i-1,j}^{K} - 2T_{i,j}^{k} + T_{i+1,j}^{K}\right)/\Delta x^2\right) + \left(T_{i,j-1}^{K} - 2T_{i,j}^{k} + T_{i,j+1}^{K}\right)/\Delta y^2\right)] \quad (9)$$

Considerando, $\Delta x^2=\Delta y^2=h$, temos que:

$$(T_{i,j}^{K+1} - T_{i,j}^{k})/\Delta t = \alpha\left[\left(T_{i,j-1}^{K} + T_{i-1,j}^{K} - 4T_{i,j}^{k} + T_{i,j+1}^{K} + T_{i+1,j}^{K}\right)/h^2\right)] \quad (10)$$

$$T_{i,j}^{K+1} = \left(1 - \frac{4\alpha\Delta t}{h^2}\right)T_{i,j}^{k} + \alpha\Delta t\left[\left(T_{i,j-1}^{K} + T_{i-1,j}^{K} - 4T_{i,j}^{k} + T_{i,j+1}^{K} + T_{i+1,j}^{K}\right)/h^2\right)] \quad (11)$$

Considerando os limites de estabilidade,

$$\left(1 - \frac{4\alpha\Delta t}{h^2}\right) \geq 0 \qquad (12)$$
$$\Delta t \geq h^2/4\alpha$$

Logo, temos que:

$$T_{i,j1}^{k+1} = (T_{i,j-1}^{K} + T_{i-1,j}^{K} + T_{i,j+1}^{K} + T_{i+1,j}^{K}) / 4 \qquad (13)$$

Onde K é o número de iterações realizadas pelo método.

2.3. OTIMIZAÇÃO POR ENXAME DE PARTÍCULAS (PSO)

Otimização por enxames de partículas (PSO) (Kennedy & Eberhart, 1995) constitui um algoritmo de otimização baseado no comportamento de enxames biológicos e em conceitos de adaptação social.

Segundo o método descrito por (Przewodowski Filho, A. et al 2017), o programa atual recebe: a saída atual simulada das doses, os valores "reais" nas posições medidas, os limites das variáveis de transformação (*K, Sx, Sy, Teta, Tpx, Tpy*), uma semente aleatória, o numero máximo de gerações, o numero de partículas, as constantes de aceleração (*C1* e *C2*), momento da partícula (*W*) e seus limites (*W_ini* e *W_max*).

Como resultado procura-se os melhores parâmetros de transformação (*K, Sx, Sy, Teta, Tpx, Tpy*), o que leva ao erro quadrático mínimo entre os valores medidos e o respectivo valor no mapa transformado.

2.3.1 CODIGO SIMPLIFICADO (PSO)

O PSO pode ser dividido em módulos e executado para todas as partículas, como vemos no trecho de código da Figura 3:

.

```
for (int g=0; g<=Max_Gen; g++) {
      for (int p=0; p< NParticles; p++) {
                  Particle[p]->eval(); // Evaluate
                  Particle[p]->update_pBest(); // Update pBest
                  Particle[p]->update_gBest(); // Update gBest
                  Particle[p]->move(); // Move particles
                  }
      }
```

Figura 3– PSO sequencial

2.3.2 EVAL (PSO)

A primeira parte da função de avaliação (Eval) constitui-se na aplicação das transformadas geométricas que como vimos na seção 2.1, resulta em uma imagem com "furos". Logo, é necessário a aplicação de um filtro para preencher os valores vazios, como vemos na Figura 4.

```
eval() = Trasnformadas() + Filtragem();
```

Figura 4 – PSO (eval)

Após essa filtragem é feito o calculo da Fitness, que corresponde à diferença quadrática entre o valor simulado para a posição e o valor transformado e filtrado calculado anteriormente, como vemos na Figura 5.

$$F(K, Sx, Sy, \emptyset, Tx, Ty) = \sqrt{(\quad (D'1 - D1)^2 + (D'2 - D2)^2 + \cdots + (D'N - DN)^2 \quad)}$$

Figura 5 – PSO (fitness)

2.3.3 UPDATE_PBEST (PSO)

A partir da fitness atual calculada, cada partícula compara se o valor calculado é menor que o valor anterior local. Caso positivo os valores de (*K, Sx, Sy, Teta, Tpx, Tpy*) são atualizados, juntamente com o valor da melhor fitness local, como vemos no trecho de código da Figura 6.

```
pBestFit = -1.7E308;            // Construtor

if (Fitness > pBestFit) {
      for (int d = 0; d < Dimension; d++) pBest[d] = X[d];
      pBestFit = Fitness;
      }
```

Figura 6 – PSO (update pBest)

2.3.4 UPDATE_GBEST (PSO)

A partir da *fitness* atual calculada, cada partícula compara se o valor calculado é menor que o valor anterior global. Caso positivo os valores de (*K, Sx, Sy, Teta, Tpx, Tpy*) são atualizados, juntamente com o valor da melhor *fitness* global, como vemos no trecho de código da Figura 7.

```
gBestFit = -1.7E308;            // Construtor

if (Fitness > gBestFit) {
      for (int d = 0; d < Dimension; d++) gBest[d] = X[d];
      gBestFit = Fitness;
      }
```

Figura 7 – PSO (update gBest)

2.3.5 MOVE (PSO)

Primeiramente é feita uma atualização no momento da partícula (*W*). Lembrando que, segundo (Waintraub,M., 2009), o uso de valores elevados de *W* ajuda a promover a exploração e prospecção globais da função, enquanto que valores baixos conduzem a uma busca local. Para que haja um equilíbrio entre essas características, é feita a inicialização de *W* com um valor alto e depois decresce (linearmente) durante a execução do programa, como vemos no trecho de código da Figura 8.

```
W = W_ini; // Construtor


dw = (2*(( W_ini - W_fim)/(Max_Gen))); // Construtor

W -= dW;
if (W<=W_fim) W = W_fim;
if (W>=W_ini) W = W_ini;
```

Figura 8 – PSO (move – atualizar W)

Em seguida são atualizadas as velocidades individuas de cada partícula, levando em conta os melhores locais, melhores globais, momento de inércia e uma variável aleatória, como vemos no trecho de código da Figura 9.

```
            for (d = 0; d < Dimension; d++) {



            double r1 = Rand();
            double r2 = Rand();

            V[d] = W*V[d] + r1*C1*(pBest[d]-X[d]) + r2*C2*(gBest[d]-X[d]);
      }
```

Figura 9 – PSO (move – atualizar V)

Lembrando que, ainda segundo (Waintraub,M., 2009), valores muito altos de velocidade podem fazer com que a partícula tente "voar" para fora do espaço de busca.

Torna-se adequado, então, uma limitação do valor da velocidade da partícula em um valor máximo (VMAX). Quando a velocidade exceder este limite, ela será fixada em VMAX, como vemos no trecho de código da Figura 10.

```
// Calculate V_MOD
      double V_MOD = 0;
      for (d = 0; d < Dimension; d++) V_MOD += powl(V[d],2.0);
      V_MOD = powl(V_MOD,0.5);

      // Limit V_MOD to V_MAX
      if (V_MOD>V_MAX)
      for (d = 0; d < Dimension; d++) V[d] = V[d] * (V_MAX/V_MOD);
```

Figura 10 – PSO (move – limitar V_mod)

Por último, são calculadas as novas partículas. Porém como vimos anteriormente existe a possibilidade de uma nova posição ser achada fora da região do problema, nesses casos é feita a reflexão da partícula para dentro do espaço de busca, como vemos no trecho de código da Figura 11.

```
	for (d = 0; d < Dimension; d++){

		X[d] = X[d] + V[d];

		// Boundary reflection
		while ((X[d]>Max[d]) || (X[d]<Min[d])) {
			if (X[d]>Max[d]) { X[d]=2*Max[d]-X[d]; V[d]=-V[d]; }
			if (X[d]<Min[d]) { X[d]=2*Min[d]-X[d]; V[d]=-V[d]; }
		}

	}
```

Figura 11 – PSO (move – atualizar partículas)

2.4 COMPUTAÇÃO PARALELA

A computação paralela desempenha um papel fundamental em pesquisas e resolução de problemas que exigem alto desempenho computacional. Uma vez que, segundo (NAVARRO & KAHLER & MATEU, 2014) a utilização de processadores simples (único núcleo) para a execução de algoritmos sequenciais pode não ser suficiente rápida para resolver problemas com maior complexidade. Expandindo essa limitação para CPU's com múltiplos núcleos, percebemos que o uso de processadores massivamente paralelos se faz cada vez mais necessário para resolução de problemas complexos em tempo hábil. Devido a essa percepção, vem se estudando e aplicando cada vez mais o uso de GPU's na resolução de tais problemas, obtendo bons resultados.

2.4.1 GPU

A GPU é um hardware projetado para alcançar um alto desempenho através de um paralelismo maciço. Para tal, sua arquitetura foi projetada de forma a garantir que a maior parte de sua superfície seja composta por Unidades Lógicas Aritméticas em detrimento das regiões de controle e cacheamento de memória, como vemos na Figura 12. Diametralmente

oposta a essa arquitetura, está a CPU, onde a maior parte de sua superfície é composta por regiões de controle e cacheamento de memória em detrimento da região das Unidades Lógicas Aritméticas, A diferença de arquiteturas e, por consequência, de desempenhos, deve-se ao propósito distinto dos dois dispositivos.

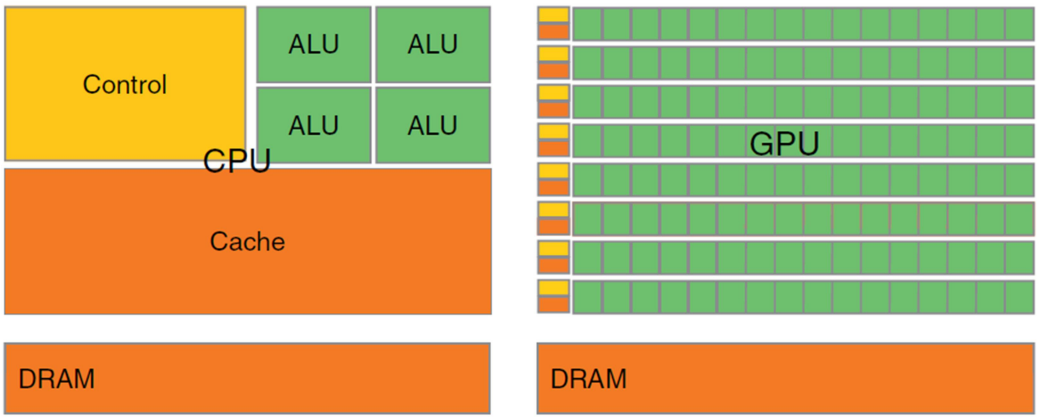

Figura 12– Arquitetura simplificada da CPU e da GPU (Carvalho Dos Santos, M. et al, 2018).

Para esse trabalho foi usada à placa de vídeo Nvidia GTX480, que como podemos ver na Tabela 2, possui as seguintes características:

Tabela 2 - Características GTX480 (Nvidia, 2018).

CUDA Cores	480
Graphics Clock (MHz)	700 MHz
Processor Clock (MHz)	1401 MHz
Texture Fill Rate (billion/sec)	42
Memory Clock (MHz)	1848
Standard Memory Config	1536 MB GDDR5
Memory Interface Width	384-bit
Memory Bandwidth (GB/sec)	177.4

2.4.2 CUDA

O uso da plataforma e modelo de programação paralela, desenvolvido pela NVIDIA, a *Compute unified device architecture* (CUDA) foi o escolhido para ser empregado no desenvolvimento deste trabalho.

O CUDA C funciona como uma extensão de linguagens C, adicionando recursos que facilitam o desenvolvimento paralelo. A seguir explicaremos como funcionam algumas de suas funções principais, que foram usadas durante o desenvolvimento do projeto

2.4.2.1 GPUERRCHK

Para simplificar o código e evitar repetições desnecessárias, usaremos uma função de checagem de erros para encapsular as funções a serem executadas no dispositivo. Como vemos na figura 13, a função recebe a operação desejada, a executa e em caso de erro, um aviso é emitido e o programa é terminado.

```
#define gpuErrchk(ans) { gpuAssert((ans), __FILE__, __LINE__); }


static inline void gpuAssert(cudaError_t code, char *file, int line, bool
abort = true) {
	if (code != cudaSuccess) {
		fprintf(stderr, "GPUassert: %s %s %d\n",
cudaGetErrorString(code), file, line);
		if (abort)	exit(code);

	}
	}
```

Figura 13 – Função GPUERRCHK

2.4.2.2 ALOCAÇÃO DE MEMÓRIA

Para usarmos a GPU, primeiramente precisamos alocar as variáveis no dispositivo, para isso usamos a função *cudaMalloc* (aloca um tamanho fixo de memória no dispositivo e

retorna seu endereço) encapsulada pela função de checagem de erros vista anteriormente. Como vemos abaixo:

```
gpuErrchk(cudaMalloc((void **)&x_dev, DIM * sizeof(int)));
```

Caso seja necessário transferir o conteúdo entre a CPU e a GPU, usamos a função *cudaMemcpy* (cópia uma tamanho fixo de bytes da memória apontada como origem para a memória apontada como destino, onde *cudaMemcpyHostToDevice* e *cudaMemcpyDeviceToHost* especificam a direção da cópia), como vemos abaixo:

```
gpuErrchk(cudaMemcpy(x_dev, x, DIM * sizeof(int), cudaMemcpyHostToDevice));

gpuErrchk(cudaMemcpy(x, x_dev, DIM * sizeof(int), cudaMemcpyDeviceToHost));
```

Por ultimo, após a execução usamos a função *cudaFree* para liberar o espaço alocado pela *cudaMalloc*, como vemos abaixo:

```
gpuErrchk(cudaFree(x_dev));
```

2.4.2.3 GERADOR RANDOMICO

Nesse trabalho vamos necessitar de números aleatórios, como vimos na descrição do PSO. Por sorte o CUDA já disponibiliza um gerador bastante eficiente, para usarmos precisamos incluir o header *curand.h* e a biblioteca dinâmica *cuRAND.*

Para usar o gerador, primeiro precisamos cria-lo, através do código:

```
curandCreateGenerator(&prngGPU, CURAND_RNG_PSEUDO_MTGP32);
```

Em seguida, definimos sua semente, como vemos:

```
curandSetPseudoRandomGeneratorSeed(prngGPU, Seed);
```

Agora geramos os valores randômicos desejados, como vemos:

```
curandGenerateUniform(prngGPU, dev_rand,sizeof(float) * 12 * NParticles);
```

Por último, é necessário liberar o gerador, como vemos abaixo:

```
curandDestroyGenerator(&prngGPU);
```

2.4.2.4 KERNEL

Chamamos de Kernel as funções executadas pela GPU. Abaixo vemos um exemplo simples, a função pso_cuda_kernel_INI_1d que recebe os ponteiros para as matrizes de entrada e saída além do seu tamanho e cópia os valores de uma variável matricial para outra, como vemos no trecho de código da Figura 14.

```
__global__ void pso_cuda_kernel_INI_1d(int NParticles, float* src, float* dst)

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

	if ((x < NParticles)) {
		dst [x] = src[x];
	}
}
```

Figura 14 – pso_cuda_kernel_INI_1d

Quando a kernel é chamada, a CPU envia uma solicitação para a GPU solicitando a criação de uma *grid* de *threads*, como vemos no trecho de código da Figura 15. Lembrando que dentro desta grid os threads são agrupados em blocos, como mostrado na Figura 16.

```
	dim3 dimBlock1a(6);
	dim3 dimGrid1a((12);

	pso_cuda_kernel_INI_1d << <dimGrid1a, dimBlock1a >> >(NParticles,
	dev_pBestFit, aux_ini);
```

Figura 15 – Chamada do kernel: `pso_cuda_kernel_INI_1d`.

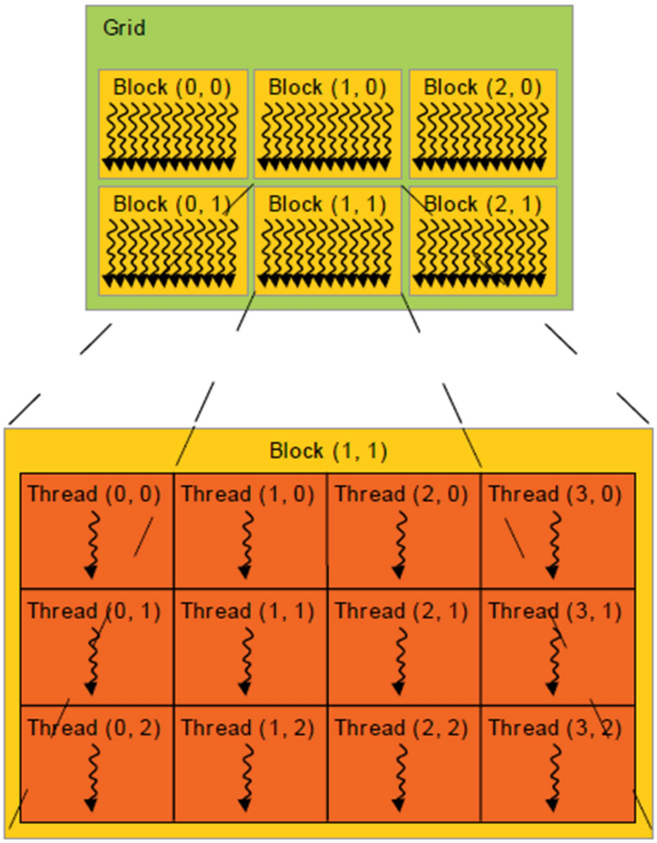

Figura 16 – Exemplo de uma grid com 6 blocos com cada um contendo 12 threads, (Carvalho Dos Santos, M. et al ,2018).

2.4.2.5 SINCRONIZAÇÃO

Devido ao uso de múltiplas funções em paralelo necessitamos usar uma função de sincronização dos módulos e evitar erros de sequencia. Para tal a biblioteca disponibiliza a função cudaDeviceSynchronize (bloqueia a execução ate que a GPU tenha terminado todas suas execuções, ou seja, servindo como uma barreira entre os blocos evitando possíveis condições de corrida) como vemos abaixo:

```
gpuErrchk(cudaDeviceSynchronize());
```

3. PROGRAMA DESENVOLVIDO

Como vimos anteriormente, mostrou-se necessário a criação de um modelo paralelo do programa (Przewodowski Filho, A. et al 2017), de forma a tornar possível o uso de resoluções melhores para a descrição da área amostrada sem estourar a janela temporal de execução total disponível (15 minutos). Para tal, foi usado GPU e CUDA, como veremos nessa secção.

Lembrando que, assim como o método descrito por (Przewodowski Filho, A. et al 2017), o programa atual recebe: a saída atual simulada das doses, os Valores "reais" nas posições medidas, os limites das variáveis de transformação (*K, Sx, Sy, Teta, Tpx, Tpy*), uma semente aleatória, o numero máximo de gerações, o numero de partículas, as constantes de aceleração (*c1 e c2*), momento da partícula (*W*) e seus limites (*W_ini* e *W_max*).

3.1. CODIGO SIMPLIFICADO (PSO-CUDA)

O PSO-CUDA pode ser dividido em módulos e executado para todas as gerações, como vemos no trecho de código da Figura 17.

```
for (int g=0; g<=Max_Gen; g++) {


            TRANSFORMAÇÃO ();
            FILTRO POR DIFUSÃO ();
            FITNESS, UPDATE PBEST E UPDATE PBEST_FIT ();
            UPDATE GBEST E GBEST_FIT ();
            MOVE();

            }
```

Figura 17 – PSO CUDA

3.2. TRANSFORMAÇÃO (PSO-CUDA)

Na versão atual do projeto, foi proposta uma forma aglutinada das transformações a ser executada em paralela em todas as partículas. Como vimos na Seção 2.1, a resultante de todas as transformações aplicadas pode ser descrita pela expressão:

$$MatrizTrasnf \begin{bmatrix} x * cos\,\theta * \mathrm{Sx} - \mathrm{y} * sen\,\theta * \mathrm{Sy} - Tx * cos\,\theta * \mathrm{Sx} + \mathrm{Ty} * sen\,\theta * \mathrm{Sy} + \mathrm{Tx} + Tpx \\ x * \mathrm{sen}\,\theta * \mathrm{Sx} + \mathrm{y} * \cos\theta * \mathrm{Sy} - Tx * sen\,\theta * \mathrm{Sx} - \mathrm{Ty} * cos\,\theta * \mathrm{Sy} + \mathrm{Ty} + Tpy \end{bmatrix}' = MatrizSim \begin{bmatrix} x \\ y \end{bmatrix}' * K \quad (14)$$

Logo, utilizando a expressão acima, podemos formular uma Função Kernell, como vemos no trecho de código da Figura 18.

```
	__global__ void pso_cuda_kernel_Transf(float *dev_Mat_resultados_, int NParticles_,
float *dev_X_, float *dev_Mat_mod_) {

	int p = blockIdx.x*blockDim.x + threadIdx.x;
	int row = blockIdx.y*blockDim.y + threadIdx.y;
	int col = blockIdx.z*blockDim.z + threadIdx.z;


	float Sx, Sy, Tx = 39.0, Ty = 16.0, Teta, Tpx, Tpy,K;
		K = dev_X[p * 6 + 0];
		Sx =dev_X[p * 6 + 1];
		Sy = dev_X[p * 6 + 2];
		Teta =dev_X[p * 6 + 3] * 3.14159 / 180.0; // conv. de teta para Radianos
		Tpx = dev_X[p * 6 + 4];
		Tpy = dev_X[p * 6 + 5];


		int xn = ((row * Sx * cos(Teta)) + (col * -Sy * sin(Teta)) + (1 * ((-
Tx*Sx*cos(Teta)) + (Ty*Sy*sin(Teta)) + Tx +Tpx)));
		int yn = ((row * Sx * sin(Teta)) + (col * Sy * cos(Teta)) + (1 * ((-
Tx*Sx*sin(Teta)) - (Ty*Sy*cos(Teta)) + Ty + Tpy)));

		if ((xn >= 0 && xn <= (NX - 1)) && (yn >= 0 && yn <= (NY - 1)) && (p <
NParticles_)) {
			dev_Mat_mod_[xn * (NY + 1) + yn + (NY + 1)*(NX + 1)*p] =
dev_Mat_resultados_[(row) *(NY + 1) + (col)+(NY + 1)*(NX + 1)*p] * K;

		}
	}
```

Figura 18 – pso_cuda_kernel_Transf.

Como podemos perceber, a função possui 3 dimensões, sendo 2 para definição da região no mapa de doses e 1 para a posição da partícula. Tendo isso em mente, por tentativa e erro e buscando sempre o menor tempo de execução, definimos o tamanho da grid usada e de seus blocos, como vemos no trecho de código da Figura 19.

```
        const int b2 = 16;
        dim3 dimBlock2a(1, b2, b2);
        dim3 dimGrid2a((NParticles) / b1, (NX + 1 + b2 - 1) / b2, (NY + 1 + b2 - 1)

/ b2);
```

Figura 19 – Definição da grid2a e de seus blocos.

Para executar a função Kernell, devemos fornecer a matriz com os valores simulados das doses, o número de partículas, a matriz com todas as partículas (cada uma composta por 6 atributos) e a matriz resultado para todas as partículas, como vemos no trecho de código da Figura 20.

```
        pso_cuda_kernel_transf    <<    <dimGrid2a,    dimBlock2a    >>    >

(dev_Mat_resultados, NParticles, dev_X, dev_Mat_mod);


        gpuErrchk(cudaDeviceSynchronize());
```

Figura 20 – Chamada da função `pso_cuda_kernel_transf`.

3.3. FILTRO POR DIFUSÃO (PSO-CUDA)

Como vimos na Seção 2.1, o modulo das transformadas geométricas tende a deixar descontinuidades no mapa calculado. No trabalho de (Przewodowski Filho, A. et al 2017) foi usado um filtro interpolador em *X* (analisa sequencialmente linha por linha) e em *Y* (analisa sequencialmente coluna por coluna) combinados, como vemos na Figura 21.

Verifica-se se houve mudança de angulo ou na escala em x (S_x ou) Encontra os espaços vazios em x Calcula-se o fator de interpolação para cada espaço Preenche os valores em x. Verifica-se se houve mudança de angulo ou na escala em x (S_y ou) Encontra os espaços vazios em y Calcula-se o fator de interpolação para cada espaço	Verifica-se se houve mudança de angulo ou na escala em x (S_y ou) Encontra os espaços vazios em y Calcula-se o fator de interpolação para cada espaço Preenche os valores em y. Verifica-se se houve mudança de angulo ou na escala em x (S_x ou) Encontra os espaços vazios em x Calcula-se o fator de interpolação para cada espaço Preenche os valores em x.

Figura 21– Interpolação XY e YX (Przewodowski Filho, A. et al 2017).

Embora esse método tenha mostrado bons resultados com bons resultados em sua forma sequencial, na versão atual do CUDA ela se mostrou muito custosa e como nosso objetivo é reduzir tempo de execução, propomos a utilização de um método computacionalmente mais simples e que pudesse ser aplicado diretamente a cada ponto.

Neste trabalho, foi proposta uma forma simplificada de filtragem por difusão a ser executada de forma paralela em todas as partículas. Como vimos na Secção 2.2, a resultante de todas as transformações aplicadas pode ser descrita pela expressão:

$$T_{i,j1}^{k+1} = (T_{i,j-1}^{K} + T_{i-1,j}^{K} + T_{i,j+1}^{K} + T_{i+1,j}^{K}) / 4 \quad (15)$$

Logo, utilizando a expressão acima podemos formular uma Função Kernell para sua execução, como vemos no trecho de código da Figura 22.

```
	__global__ void pso_cuda_kernel_DIF(int NParticles, float* dev_Mat_mod_, float
*dev_Mat_mod_Media_)

{
	int pos = blockIdx.x*blockDim.x + threadIdx.x;
	int x = blockIdx.y*blockDim.y + threadIdx.y;
	int y = blockIdx.z*blockDim.z + threadIdx.z;

	if ((y <= NY )  && (pos<NParticles)  && (x <= NX )) {

		if ((x == 0) || (x == (NX + 1)) || (y == 0) || (y == (NY + 1)) )
			dev_Mat_mod_Media_[x*(NY + 1) + y + (NX + 1)*(NY + 1)*pos] =
dev_Mat_mod_[x*(NY + 1) + y + (NX + 1)*(NY + 1)*pos];

		else
		{

			dev_Mat_mod_Media_[x*(NY + 1) + y + (NX + 1)*(NY + 1)*pos] = 0.25*
				(dev_Mat_mod_[(x - 1)*(NY + 1) + y + (NX + 1)*(NY + 1)*pos]
				+ dev_Mat_mod_[(x + 1)*(NY + 1) + y + (NX + 1)*(NY + 1)*pos]
				+ dev_Mat_mod_[x*(NY + 1) + y - 1 + (NX + 1)*(NY + 1)*pos]
				+ dev_Mat_mod_[x*(NY + 1) + y + 1 + (NX + 1)*(NY + 1)*pos]);
		}
	}
	}
```

Figura 22 –Função pso_cuda_kernel_DIF.

Como podemos perceber, essa função, assim como sua predecessora, utiliza 3 dimensões para descrever suas regiões. Logo, usaremos o mesmo tamanho de grid e blocos que anteriormente, descrito na Figura 19.

Para executar a função Kernell, devemos fornecer a matriz com os valores transformados das doses (obtido na etapa anterior), o número de partículas e a matriz resultado para todas as partículas. Lembrando que a função é iterativa e empiricamente definimos o numero de iterações (*K*) para 2, logo temos que executar múltiplos Kernell's seguidos pela função de sincronização, como vemos no trecho de código da Figura 23.

```
        pso_cuda_kernel_DIF  <<  <dimGrid2a,  dimBlock2a  >>  >  (NParticles,
dev_Mat_mod, dev_Mat_mod_Media);

        gpuErrchk(cudaDeviceSynchronize());

        pso_cuda_kernel_DIF  <<  <dimGrid2a,  dimBlock2a  >>  >  (NParticles,
dev_Mat_mod_Media, dev_Mat_mod);

        gpuErrchk(cudaDeviceSynchronize());
```

Figura 23 – Chamada da função pso_cuda_kernel_DIF.

3.4. FITNESS, UPDATE PBEST E UPDATE PBEST_FIT (PSO-CUDA)

Neste trabalho, foi proposta uma forma aglutinada entre a *fitness* e *update_gBest* a ser executada de forma paralela em todas as partículas. Como vimos na Figura 5 e Figura 6.

Essa aglutinação se justifica, na medida em que a primeira função calcula a fitness para atualizar seu valor, e a segunda necessita utilizar esse resultado para atualizar as variáveis locais de cada partícula (*pBestFit* e *pBest*). Logo, chamar 2 kernell's diferentes e sincroniza-los, nos custaria mais tempo de execução do que as aglutinarmos numa Função Kernell unica para sua execução, como vemos no trecho de código da Figura 24.

```
__global__ void pso_cuda_kernel_8(int *x_dev_, int *y_dev_, float
*v_med_dev_,int NParticles_, float *dev_X_, float *dev_Mat_mod_, float
*dev_fitnessResult_,float *dev_pBestFit_, float *dev_pBest_, float *dev_gBestFit_,
float *dev_gBest_) {


	int pos = blockIdx.x*blockDim.x + threadIdx.x;
	float soma = 0.0, prod;

	if (pos < NParticles_) {

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

			prod = v_med_dev_[i] - dev_Mat_mod_[(x_dev_[i] - 1)* (NY + 1) +
			y_dev_[i] - 1 + (NX + 1)*(NY + 1)*pos];
			soma += (prod*prod);
		}

		dev_fitnessResult_[pos] = -(soma / (float)(DIM));

		//////////Update pbest, pbestFit ////////////////////////

		if (dev_fitnessResult_[pos] > dev_pBestFit_[pos]) {
			dev_pBestFit_[pos] = dev_fitnessResult_[pos];

			for (int i = 0; i < 6; i++)
				dev_pBest_[pos * 6 + i] = dev_X_[pos * 6 + i];
		}
	}
}
```

Figura 24 – Função pso_cuda_kernel_8.

Como podemos perceber, a função possui 1 dimensões para a controlar a posição da partícula. Tendo isso em mente, por tentativa e erro e buscando sempre o menor tempo de execução, definimos o tamanho da grid usada e de seus blocos, como vemos no trecho de código da Figura 25.

```
	const int b1 = 4;
	dim3 dimBlock1a(b1);
	dim3 dimGrid1a((NParticles + b1 - 1) / b1);
```

Figura 25 – Definição da grid1a e de seus blocos.

Para executar a função Kernell, devemos fornecer: a matriz com os valores “reais” das doses; as matrizes com sua posição *X* e *Y*; o número de partículas; a matriz com todas as partículas (cada uma composta por 6 atributos); a matriz resultado para todas as *fitness* das partículas; a matriz com todas as partículas *p_Best* (cada uma composta por 6 atributos) e a variável *pBestFit,* como vemos no trecho de código da Figura 26.

```
	pso_cuda_kernel_8 << <dimGrid1a, dimBlock1a >> > (x_dev, y_dev,
v_med_dev, NParticles, dev_X, dev_Mat_mod, dev_fitnessResult,dev_pBestFit,
dev_pBest);

	gpuErrchk(cudaDeviceSynchronize());
```

Figura 26 – Chamada da função pso_cuda_kernel_8.

3.5. UPDATE GBEST E GBEST_FIT (PSO-CUDA)

Como vimos na Seção 2.33, uma vez que a *fitness* é calculada, cada partícula compara se o valor calculado é menor que o valor global obtido nas iterações anteriores. Caso positivo os valores de (*K, Sx, Sy, Teta, Tpx, Tpy*) são atualizados, juntamente com o valor da melhor *fitness* global e a posição da melhor partícula é guardada na variável de controle, como vemos no trecho de código da Figura 27.

```
__global__ void pso_cuda_gbest_gbestfit(int NParticles_, float *dev_X_,
float *dev_fitnessResult_,
float *dev_gBestFit_, float *dev_gBest_,int *dev_particula_best_) {

int particula_best_;
for (int i = 0; i < NParticles_; i++) {

	if ((dev_fitnessResult_[i] > *dev_gBestFit_)) {
		*dev_gBestFit_ = dev_fitnessResult_[i];
		particula_best_ = i;
		*dev_particula_best_ = particula_best_;
		for (int j = 0; j < 6; j++) {
			dev_gBest_[j] = dev_X_[particula_best_ * 6 + j];
			}
		}
	}

}
```

Figura 27 – Função pso_cuda_gbest_gbestfit.

Como podemos perceber a função simplesmente varre a matriz com os resultados da *fitness* de cada partícula buscando o melhor resultado. Logo, não há necessidade de subdividir o domínio da função em múltiplos blocos de execução.

Para executar a função Kernell, devemos fornecer: o número de partículas; a matriz com todas as partículas (cada uma composta por 6 atributos); a matriz resultado para todas as *fitness* das partículas; a matriz com as partículas *g_Best* (cada uma composta por 6 atributos) e a variável *gBestFit*, como vemos no trecho de código da Figura 28.

```
pso_cuda_gbest_gbestfit << <1,1>> > ( NParticles, dev_X,
dev_fitnessResult,dev_gBestFit, dev_gBest,dev_particula_best);

gpuErrchk(cudaDeviceSynchronize());
```

Figura 28 – Chamada da função pso_cuda_gbest_gbestfit.

3.6. MOVE (PSO-CUDA)

Como vimos na Seção 2.3.4, primeiramente é feita uma atualização no momento da partícula (W). Em seguida são atualizadas as velocidades individuas de cada partícula, levando em conta os melhores locais, melhores globais, momento de inércia e uma variável aleatória. Por último, são calculadas as novas partículas, como vemos no trecho de código da Figura 29.

```
	__global__ void pso_cuda_kernel_Move(int Dimension, float *dev_min_, float *dev_max_,
float dev_V_MAX_, float *dev_W_, float *dev_rand_, float c1, float c2, float wini, float
wfim, float dwini, int NParticles_, float *dev_X_, float *dev_V_, float *dev_gBest_, float
*dev_pBest_) {

	int pos = blockIdx.x*blockDim.x + threadIdx.x;
	if (pos < NParticles_) {

		int d;
		dev_W_[pos] = dev_W_[pos] - dwini;
		if (dev_W_[pos] <= wfim) dev_W_[pos] = wfim;
		if (dev_W_[pos] >= wini) dev_W_[pos] = wini;


		for (d = 0; d < Dimension; d++) {

			float r1 =   dev_rand_[2 * d + pos*Dimension * 2];
			float r2 =   dev_rand_[2 * d + 1 + pos*Dimension * 2];

			dev_V_[d + pos * 6] = dev_W_[pos] *dev_V_[d + pos * 6] +
r1*c1*(dev_pBest_[d + pos * 6] - dev_X_[d + pos * 6]) + r2*c2*(dev_gBest_[d] - dev_X_[d + pos
* 6]);
		}

		// Calculate V_MOD
		float V_MOD = 0;
		for (d = 0; d < Dimension; d++) V_MOD += (dev_V_[d + pos * 6]* dev_V_[d + pos *
6]);
		V_MOD = sqrt(V_MOD);

		//// Limit V_MOD to V_MAX
		if (V_MOD > dev_V_MAX_)
			for (d = 0; d < Dimension; d++) dev_V_[d + pos * 6] = dev_V_[d + pos * 6]
* (dev_V_MAX_ / V_MOD);

		//// Update position
		for (d = 0; d < Dimension; d++) {
			dev_X_[d + pos * 6] = dev_X_[d + pos * 6] + dev_V_[d + pos * 6];

			// Boundary reflection
			while ((dev_X_[d + pos * 6] > dev_max_[d]) || (dev_X_[d + pos * 6] <
dev_min_[d])) {
				if (dev_X_[d + pos * 6] > dev_max_[d]) {
					dev_X_[d + pos * 6] = 2 * dev_max_[d] - dev_X_[d + pos *
6];
					dev_V_[d + pos * 6] = -dev_V_[d + pos * 6];
				}

				if (dev_X_[d + pos * 6] < dev_min_[d]) {
					dev_X_[d + pos * 6] = 2 * dev_min_[d] - dev_X_[d + pos *
6];
					dev_V_[d + pos * 6] = -dev_V_[d + pos * 6]; }
			}
		}
	}
}
```

Figura 29 – Função pso_cuda_kernel_Move.

Como podemos perceber, a função possui 1 dimensão, para a controlar a posição da partícula. Logo, usaremos o mesmo tamanho de grid e blocos que anteriormente foi descrito na Figura 24.

Para executar a função Kernell, devemos fornecer: o numero de dimensões usadas; a matriz com os valores mínimos de cada dimensão; a matriz com os valores máximos de cada dimensão; o valor limite da velocidade da partícula; o momento da partícula e seus limites e variação incremental; a matriz com os números aleatórios; as constantes de aceleração; número de partículas; a matriz com todas as partículas (cada uma composta por 6 atributos); a matriz com todas as velocidades das partículas; a matriz com as partículas *g_Best* (cada uma composta por 6 atributos) e a matriz com as partículas p_*Best* (cada uma composta por 6 atributos), como vemos no trecho de código da Figura 30.

```
      pso_cuda_kernel_Move << <dimGrid1a, dimBlock1a >> > (Dim, dev_min,
dev_max, aux_V_MAX, dev_W, dev_rand, C1, C2, W_INI, W_FIM,
dw_ini,NParticles, dev_X, dev_V, dev_gBest, dev_pBest);



      gpuErrchk(cudaDeviceSynchronize());
```

Figura 30 – Chamada da função pso_cuda_kernel_Move.

4. TESTES E RESULTADOS

Para os testes apresentados a seguir, tanto CPU quanto GPU, foi usado um computador intel®Core i7- 3.2GHZ com 8GB de memória *ram* e uma placa de vídeo *Nvdia GeForce GTX 480* utilizando o editor *Microsoft Visual Studio 2012* e a biblioteca *Nvidia CUDA 8.0*.

Para avaliar o método proposto, foram criadas situações hipotéticas utilizando os dados gerados pelo simulador (SDAR) da Central Nuclear Almirante Álvaro Alberto (CNAAA).

Foi usado PSO com 400 gerações e 100 partículas em ambos os casos.

4.1. SITUAÇÃO 1

Nessa situação, foi usado um mapa de distribuição de doses calculado pelo SDAR como a referência estimada inicialmente (informação disponível ao operador durante o acidente), e outra simulação também usando o SDAR considerando um termo fonte e condições meteorológicas diferentes da outra para simular os dados obtidos pela equipe de emergência em tempo real, doravante chamada de referência real.

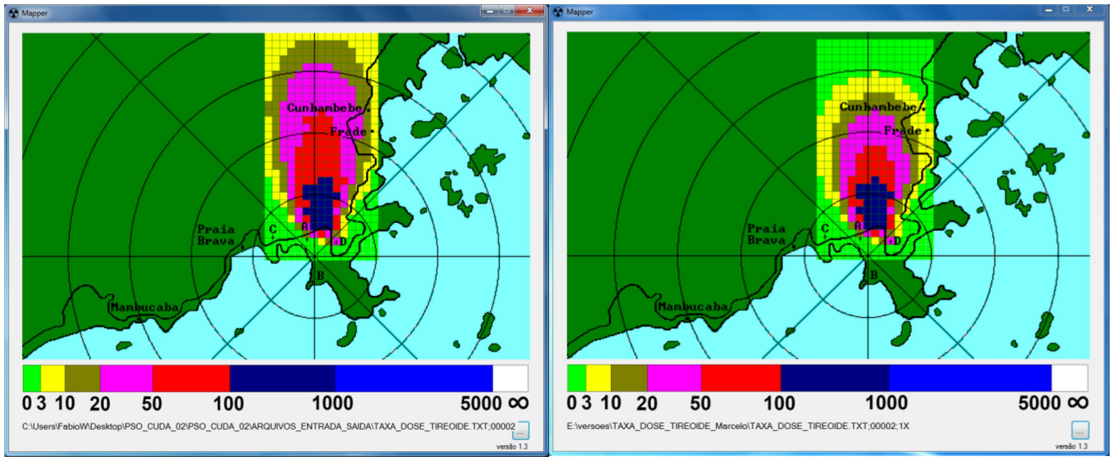

Figura 31 - Esquerda (Pluma simulada SDAR 1) e Direita (Pluma SDAR real 1)

Tabela 3- Doses de Referência 1 e erro.

Ponto(x, y)	**Pluma estimada SDAR(mRem/h)**	**Pluma "real" SDAR(mRem/h)**	**((Real-Simulado)/Real) *100**
(16,39)	1,3023	1,3023	0
(18,37)	15,1726	15,1726	0
(18,39)	114,0772	114,0772	0
(18,41)	305,0941	305,0941	0
(21,37)	53,7342	53,4888	-0,458787634
(21,39)	236,1676	235,7667	-0,170040977
(21,41)	111,4671	111,0812	-0,347403521
(24,36)	34,0855	31,9004	-6,84975737
(24,39)	99,2065	94,58587	-4,885116561
(24,42)	9,0115	75,9324	**88,13220707**
(28,34)	13,3390	8,4626	**-57,62295276**
(28,39)	78,4812	58,8623	-33,33016209
(28,44)	28,8269	19,6486	-46,71223395
(34,33)	10,1278	2,2688	**-346,3945698**
(34,39)	45,5512	14,7867	**-208,0552118**
(34,45)	18,0460	4,9203	**-266,7662541**
(40,33)	6,1801	0,2068	**-2888,44294**
(40,39)	19,3501	0,8955	**-2060,815187**
(40,45)	9,2817	0,3721	**-2394,410105**
			|8.403,33|

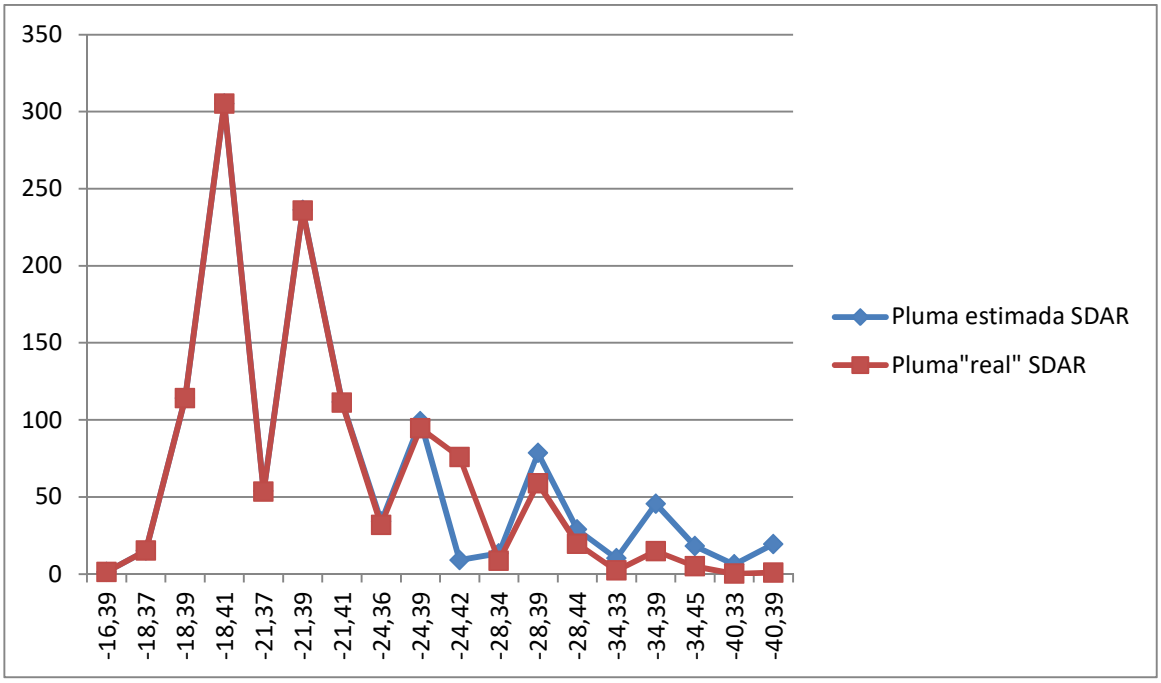

Figura 32 - Gráfico comparativo Referência Real 1 e Simulada 1.

4.1.1. *CPU X GPU*

Como vemos na Tabela 4, no caso da execução CPU, o ponto (24,42) ficou abaixo do valor “real” enquanto o ponto (34,39) ficou acima do “real”. No caso da execução GPU, os pontos (24,42), (21,39) e (18,41) ficaram abaixo do valor “real”, já o ponto (21,41) ficou acima do esperado.

Como podemos ver na Tabela 4, ambos os testes encontraram falhas pontuias porém foram capazes de chegar a resultados semelhantes ao “real” (Figura 32), no sentido que ambos levaram a resultados proximos ao “real” além de reduzir consideravelmente o erro inicial, sendo que a saida CPU mostrou-se coerente em relação à distribuiçao interna das doses da figura “real” porém obteve uma area de espalhamento um pouco superior a esta. Já na saida GPU, a distribuição de doses perto do reator foi um pouco superior a “real” porém sua área de espalhamento total ficou circuscrita a esta.

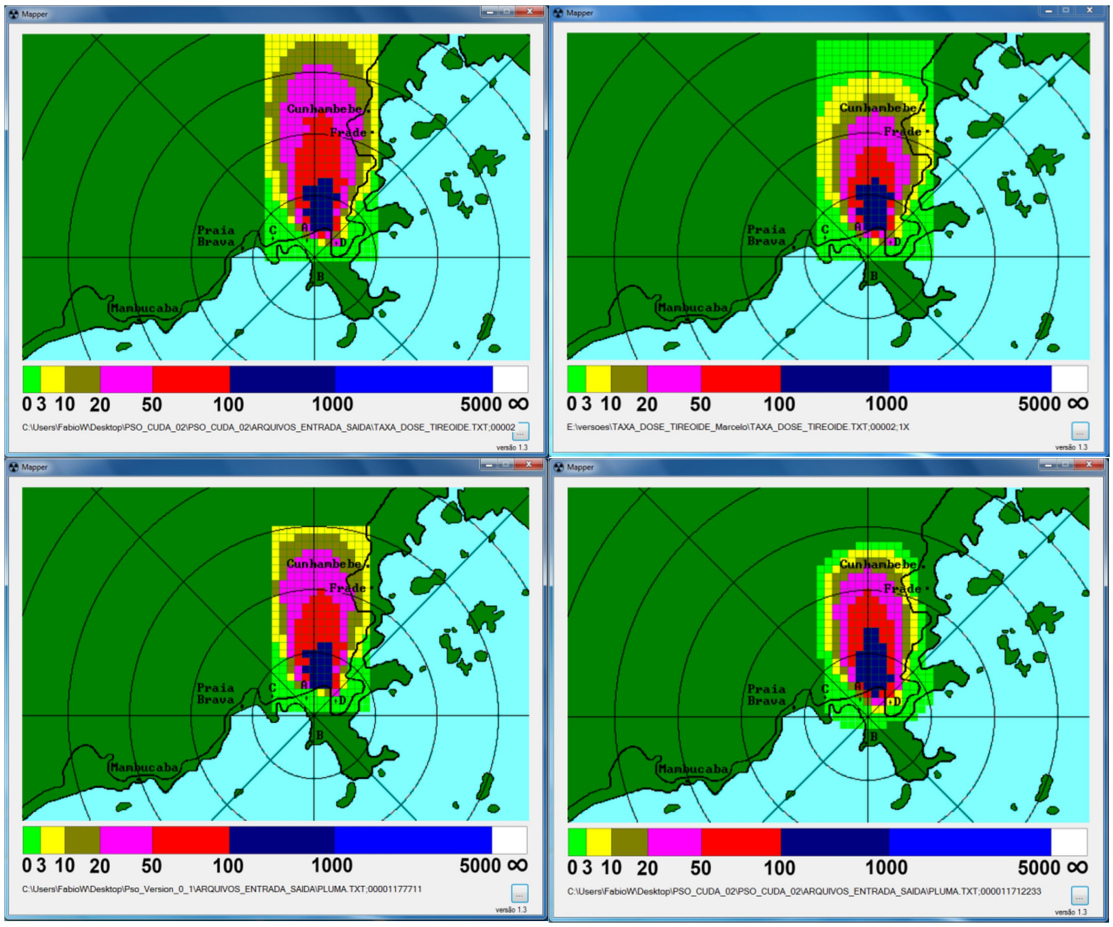

Figura 33 – Superior esquerda (referência simulada SDAR 1), superior direita (referência SDAR real 1), inferior esquerda (Saida CPU 1) e inferior direita (saída GPU 1).

Tabela 4 - Doses de Referência 1, Saídas CPU 1 e GPU 1 e erros.

Ponto (x, y)	Pluma estimada SDAR (mRem/h)	Pluma "real" SDAR (mRem/h)	Saída CPU (mRem/h)	((Real-CPU) / Real) *100	Saída GPU (mRem/h)	((Real-GPU)/Real) *100
(16,39)	1,3023	1,3023	0,1598	87,72940183	14,2215	-992,0294863
(18,37)	15,1726	15,1726	15,1228	0,328223245	15,4446	-1,792705271
(18,39)	114,0772	114,0772	113,7024	0,328549438	173,9221	-52,46000077
(18,41)	305,0941	305,0941	304,0918	0,328521594	261,4956	14,29018129
(21,37)	53,7342	53,4888	53,5576	-0,128625058	54,0975	-1,137995244
(21,39)	236,1676	235,7667	235,3917	0,159055541	193,8517	17,77816799
(21,41)	111,4671	111,0812	111,1009	-0,017734774	171,1262	-54,05505162
(24,36)	34,0855	31,9004	37,0588	-16,17033015	33,2361	-4,187094833
(24,39)	99,2065	94,58587	98,8375	-4,494994866	113,4871	-19,98314336
(24,42)	9,0115	75,9324	0,0	100	0,0	100
(28,34)	13,3390	8,4626	9,1395	-7,998723796	7,2974	13,76881809
(28,39)	78,4812	58,8623	61,0768	-3,762170354	73,1498	-24,27275183
(28,44)	28,8269	19,6486	20,2562	-3,092332278	13,1092	33,28176053
(34,33)	10,1278	2,2688	0,0	100	0,0	100
(34,39)	45,5512	14,7867	26,3676	-78,31970622	11,9980	18,85951565
(34,45)	18,0460	4,9203	9,0333	-83,59246387	0,3982	91,90699754
(40,33)	6,1801	0,2068	0,0	100	0,0	100
(40,39)	19,3501	0,8955	0,0	100	0,0	100
(40,45)	9,2817	0,3721	0,0	100	0,0	100
				|786,36|		|1.839,73|

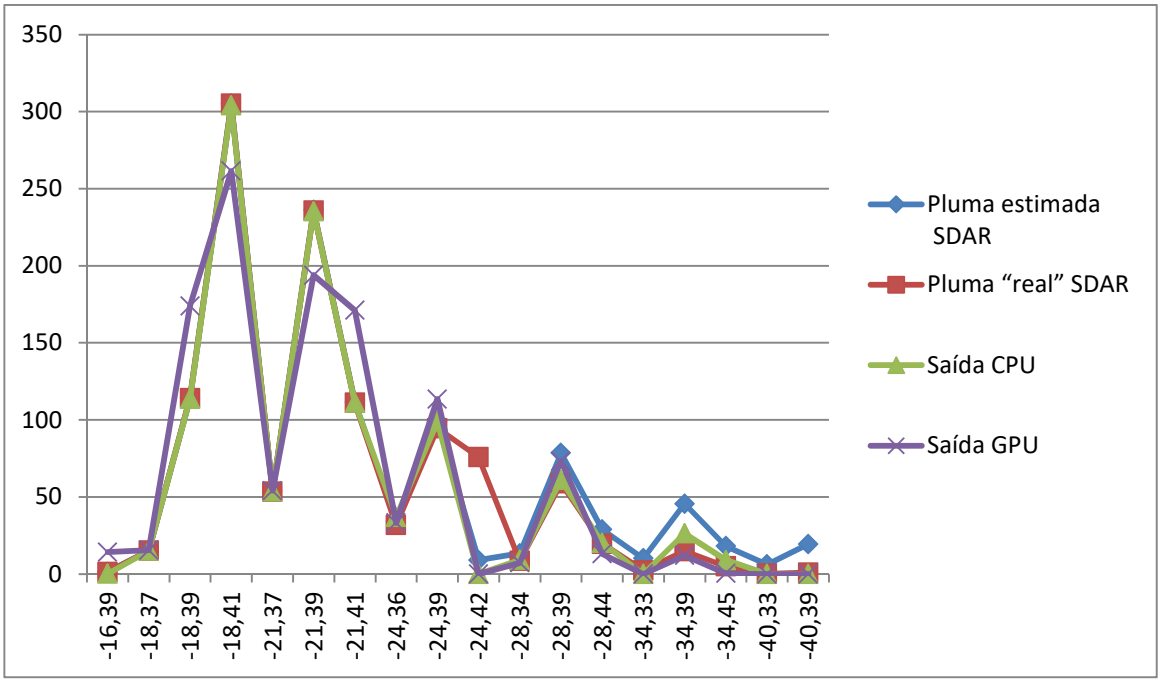

Figura 34 - Gráfico comparativo Referência Real 1, Simulada 1, Saídas CPU 1 e GPU 1.

4.2. SITUAÇÃO 2

Nessa situação, foi usado um mapa de distribuição de doses calculado pelo SDAR como a referência simulada inicial (informação disponível ao operador durante o acidente), e outra simulação também usando o SDAR considerando um termo fonte e condições meteorológicas diferentes da outra como sendo dados obtidos pela equipe de emergência em tempo real.

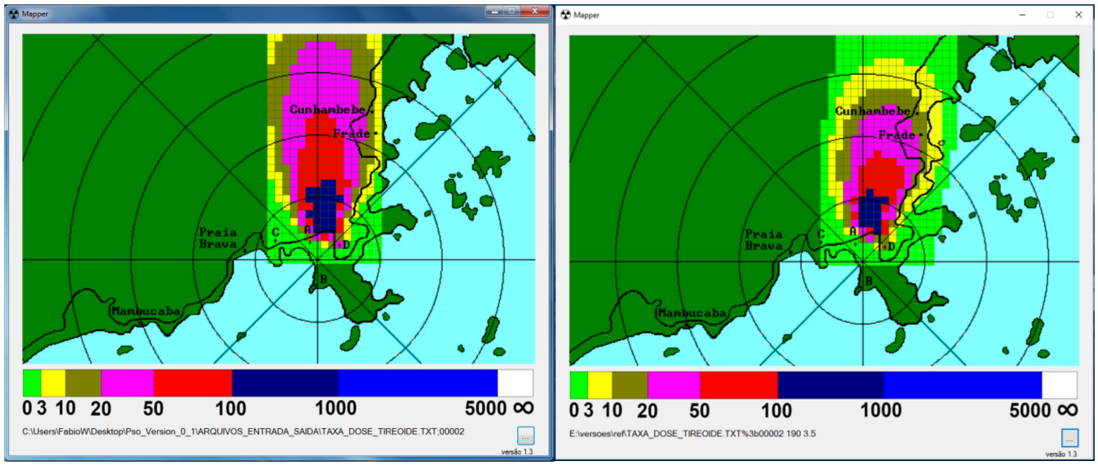

Figura 35 - Esquerda (Pluma simulada SDAR 2) e Direita (Pluma SDAR real 2)

Tabela 5 - Doses de Referência 2.

Ponto (x, y)	Pluma estimada SDAR (mRem/h)	Pluma "real" SDAR (mRem/h)	((Real- Simulado)/Real) *100
(16,39)	1,3023	0,3080	**-322,8246753**
(18,37)	15,1726	4,8450	**-213,1599587**
(18,39)	114,0772	38,9030	**-193,234969**
(18,41)	305,0941	210,5410	-44,90959006
(21,37)	53,7342	19,1431	**-180,6974837**
(21,39)	236,1676	138,606	**-70,3877177**
(21,41)	111,4671	102,7840	-8,447910181
(24,36)	34,0855	9,9545	**-242,4129791**
(24,39)	99,2065	53,8910	**-84,08732441**
(24,42)	16,2737	86,9960	**81,29373764**
(28,34)	13,3390	0,0770	**-17223,37662**
(28,39)	78,4812	40,3110	**-94,68929076**
(28,44)	28,8269	39,2670	26,5874653
(34,33)	10,8776	0,0	**#DIV/0!**
(34,39)	47,3897	13,915	**-240,5655767**
(34,45)	19,0817	15,4510	-23,49815546
(40,33)	9,5894	0,0	**#DIV/0!**
(40,39)	27,5086	2,178	**-1163,02112**
(40,45)	13,8506	2,9230	**-373,8487855**
			|20.586,94|

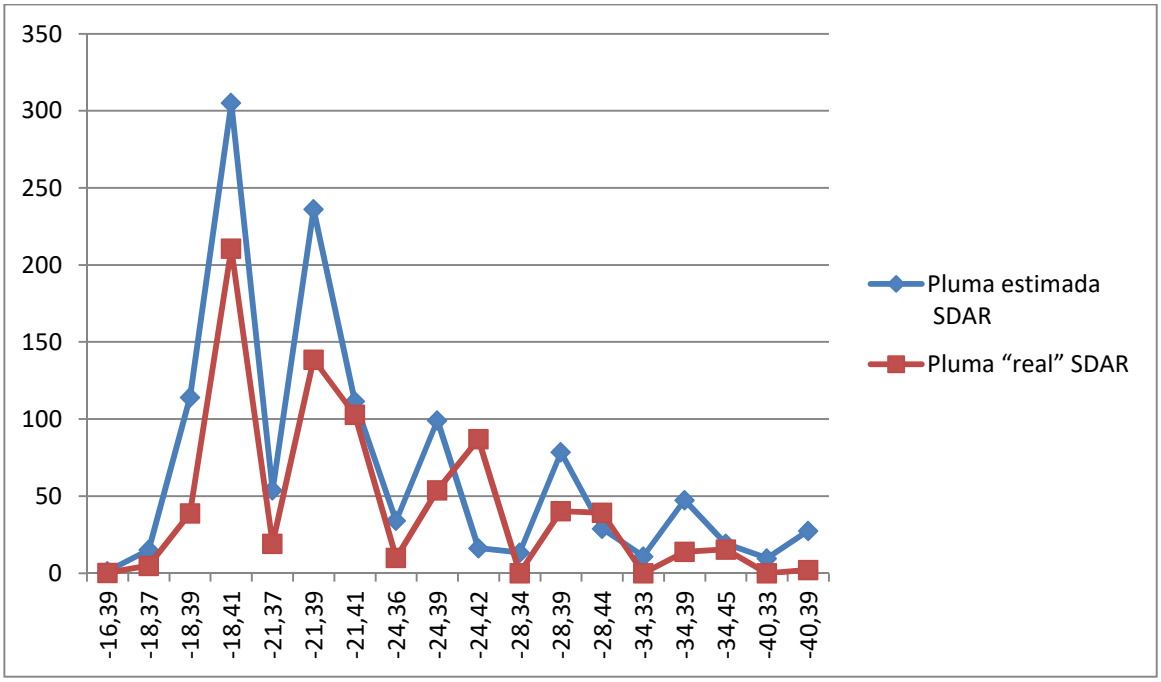

Figura 36 - Gráfico comparativo Referência Real 2 e Simulada 2.

4.2.1. *CPU X GPU*

Como vemos na Tabela 6, no caso da execução CPU, os pontos (24,42) e (21,39) ficaram abaixo do valor "real", já o ponto (21,41) ficou acima do esperado. No caso da execução GPU, os pontos (24,42) e (21,39) ficaram abaixo do valor "real", já o ponto (21,41) ficou acima do esperado.

Como vimos Tabela 6, ambos os testes encontraram falhas pontuias porém foram capazes de chegar a resultados semelhantes ao "real" (Figura 36), no sentido que ambos levaram a resultados proximos ao "real" além de reduzir consideravelmente o erro inicial, sendo que a saida CPU mostrou-se coerente em relação à distribuiçao interna das doses da figura "real" porém obteve uma area de espalhamento um pouco superior a esta. Já na saida

GPU, a distribuição de doses perto do reator foi um pouco superior à “real”, porém sua area de espalhamento total ficou circuscrita a esta.

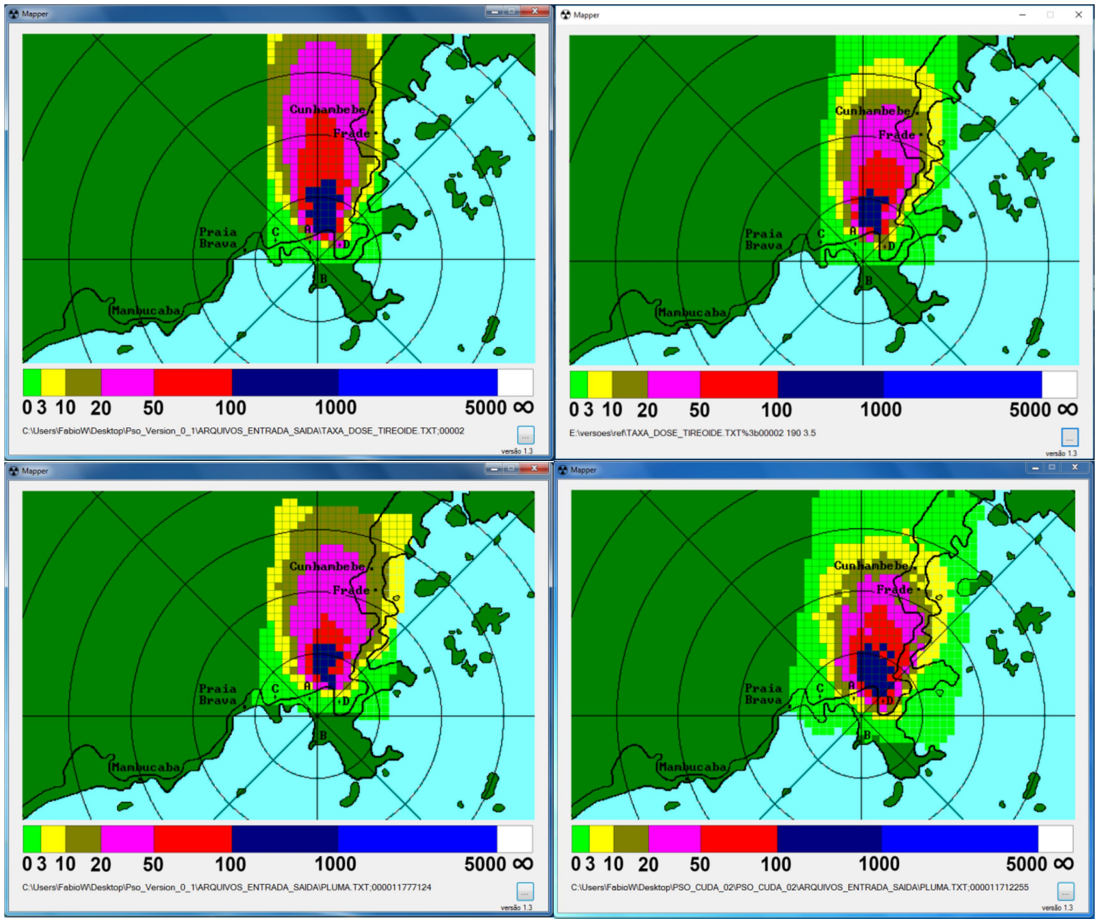

Figura 37 – Superior esquerda (referência simulada SDAR 2), superior direita (referência real SDAR 2), inferior esquerda (Saída CPU 2) e inferior direita (Saída GPU 2).

Tabela 6 - Doses de Referência 2, Saídas CPU 2 e GPU 2.

Ponto (x, y)	Pluma estimada SDAR (mRem/h)	Pluma "real" SDAR (mRem/h)	Saída CPU (mRem/h)	((Real-CPU)/ Real) *100	Saída GPU (mRem/ h)	((Real-GPU)/Real)*100
(16,39)	1,3023	0,3080	0,1020	**66,88311688**	6,7228	**-2082,727273**
(18,37)	15,1726	4,8450	4,7811	1,318885449	10,1617	**-109,7358101**
(18,39)	114,0772	38,9030	31,4298	19,20982958	64,6260	**-66,12086471**
(18,41)	305,0941	210,5410	194,2188	7,752504263	205,2235	2,525636337
(21,37)	53,7342	19,1431	31,3916	**-63,98388976**	20,8008	-8,659517006
(21,39)	236,1676	138,606	150,3411	-8,466516601	114,8028	17,17328254
(21,41)	111,4671	102,7840	115,0339	-11,91810009	119,8915	-16,64412749
(24,36)	34,0855	9,9545	19,0176	**-91,04525591**	19,5557	**-96,45085137**
(24,39)	99,2065	53,8910	63,1535	-17,18747101	65,0783	-20,7591249
(24,42)	16,2737	86,9960	6,2296	**92,839211**	0,1251	**99,85620029**
(28,34)	13,3390	0,0770	8,4914	**-10927,79221**	5,6987	**-7300,909091**
(28,39)	78,4812	40,3110	41,8700	-3,867430726	38,1852	5,273498549
(28,44)	28,8269	39,2670	33,4188	14,89342196	34,8268	11,30771386
(34,33)	10,8776	0,0	0,0	**#DIV/0!**	0,4448	**#DIV/0!**
(34,39)	47,3897	13,915	23,7712	**-70,83147682**	11,5052	17,31800216
(34,45)	19,0817	15,4510	18,7411	-21,29376739	9,7602	36,83127306
(40,33)	9,5894	0,0	0,0	**#DIV/0!**	0,0253	**#DIV/0!**
(40,39)	27,5086	2,178	11,9194	**-447,2635445**	1,0180	**53,25987144**
(40,45)	13,8506	2,9230	11,3429	**-288,056791**	0,9887	**66,1751625**
				|12.154,151|		**|10.011,63|**

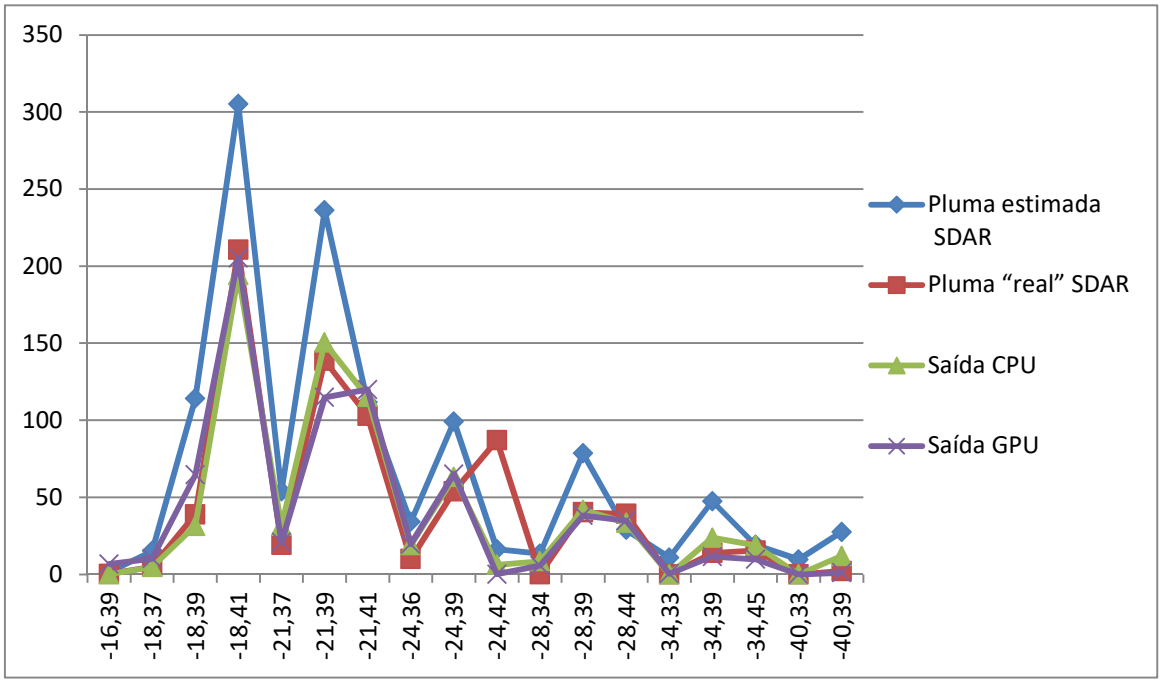

Figura 38 - Gráfico comparativo Referências Real 2, Simulada 2, Saídas CPU 2 e GPU 2.

4.3. TEMPOS DE EXECUÇÃO

Agora analisaremos a variação dos tempos de execução dos métodos quando mudamos a resolução do mapa usado para descrever o ambiente, lembrando que para ambos os testes foi usada a mesma máquina.

Tabela 7 - Comparação dos Tempos de Execução CPU/GPU.

Resolução do mapa	Tempo Execução Cpu (Δt)	Tempo Execução Gpu (Δt)	Ganho CPU/GPU
(67 x 43)	00:07 min / 7 s	0,356 s	19,66
(335 x 215)	03:02 mim / 182 s	2,833 s	64,24
(670 x 430)	19:50 mim / 1190 s	9,972 s	119,33
(938 x 602)	43:48 min / 2627,98 s	19,234 s	136,63
(1005 x 645)	50:19 min / 3019 s	---	---

5. CONCLUSÕES

Neste trabalho, foi proposta uma abordagem paralela para melhorar a estimativa de dose devido a acidentes em usinas nucleares com liberação de material radioativo para atmosfera. Assim como o método anterior, o método aqui proposto não tem como objetivo corrigir ou fazer novas estimativas para o termo fonte, a ideia é fazer a correção diretamente no mapa de distribuição de dose (usando a saída do SDAR), que é a base para a tomada de decisão sob situações severas.

Como vimos na Seção 4.1 e 4.2, a abordagem proposta aqui (GPU) demonstrou ser eficaz para corrigir o mapa de dose estimada SDAR, seja em situações onde as doses foram subdimensionadas e superdimensionadas. Além disso, ambos os métodos CPU e GPU falharam no ponto (24,42), o que nos leva a crer que talvez o método possa ser melhorado de forma a flexibilizar um pouco mais a transformação da pluma, o que poderá ser investigado futuramente.

Finalmente, analisando a Seção 4.3, percebemos que, para malha inicial de (67x43), o método GPU apresenta ganho de 19,66 vezes em relação ao tempo de execução dos sistemas CPU/GPU e esse ganho só cresce quando aumentamos a resolução do mapa usado pelo sistema, o que nos permite não só aumentar a resolução usada, mas também o número de execuções possíveis de serem feitas pelo operador do sistema dentro da janela temporal definida de 15 minutos, permitindo a aplicação prática do método ao problema real, conforme o objetivo inicial dessa dissertação.

Futuramente, pretende-se investigar o uso do filtro de difusão com múltiplas iterações; outras formas de avaliação (*fitness*) além da raiz quadrada do erro quadrático; verificar os pontos de medição com maior influência para o acerto do mapa final; validar o número ideal de rodadas do novo método; pesquisar formulas simplificadas de interpolação que possam ser paralelizadas e testar as funcionalidades implementadas no CUDA 10.0.

6. REFERÊNCIAS

INTERNATIONAL ATOMIC ENERGY AGENCY. Eletricity Information: Overview (2017 edition). p. 3. 2017.

McKenna, T. J., & Giitter, J. G. (1988). Source Term Estimation During Incident Response to Severe Nuclear Power Plant Accidents. NUREG-1228.

CALIFORNIA AIR RESOURCES BOARD AND THE CALIFORNIA ENERGY RESOURCES CONSERVATION AND DEVELOPMENT COMMISSION. Point Source Model Evaluation and Development Study – The Grid Model IMPACT (Integrated Model for Plumes and Atmospherics in Complex Terrain). FABRICK, A.; SKLAREW, R.; WILSON, J. Technical report (Contract A5-058-87). [S.l.]: [s.n.], 1987.

Carvalho Dos Santos, M. et al (2018), Modelo Computacional Paralelo Baseado Em GPU Para Cálculo Em Tempo Real Da Dispersão Atmosférica De Radionuclídeos Nas Vizinhanças De Uma Central Nuclear. PPGIEN-2018.

Przewodowski Filho, A et al (2017), Um Novo Método Para Aumentar A Acurácia Das Estimativas De Dose Devido A Acidentes Nucleares Baseado Em Medidas De Campo E Otimização Por Enxame De Particulás. Programa de Pós-graduação em Engenharia Nuclear, COPPE, da Universidade Federal do Rio de Janeiro 2017.

Athey, G. F., Brandon, L., & Jr., R. (2013). J.V. NRC. RASCAL 4.3 WORKBOOK.

Zheng, X., & Chen, Z. (2011, 1 6). Inverse calculation approaches for source determination in hazardous chemical releases. Journal of Loss Prevention in the Process Industries, pp. 293–301.

Chow, F. K., Kosović, B., & Chan, S. (2008). Source Inversion for Contaminant Plume Dispersion in Urban Environments Using Building-Resolving Simulations. American Meteorological Society, 47, 1553–1572.

Kennedy, J., & Eberhart, R. (1995). Particle Swarm Optimization. Proceedings of IEEE International Conference on Neural Networks, Vol. 4, pp. 1942-1948.

Waintraub, M. (2009). ALGORITMOS PARALELOS DE OTIMIZAÇÃO POR ENXAME DE PARTÍCULAS. UFRJ/ COPPE/ Programa de Engenharia, 7-14.

NAVARRO, C. A.; -KAHLER, N. H; MATEU, L. A survey on parallel computing and its applications in data-parallel problems using GPU architectures. Communications in Computational Physics, 15 (2), p. 285-329. 2014.

Nvidia. https://www.nvidia.com.br/object/product_geforce_gtx_480_br.html, 2018.