2017 International Nuclear Atlantic Conference - INAC 2017
Belo Horizonte, MG, Brazil, October 22-27, 2017
ASSOCIAÇÃO BRASILEIRA DE ENERGIA NUCLEAR – ABEN

Agência Brasileira do ISBN
ISBN 978-85-99141-07-6
9 788599 141076

AN OPTIMIZATION OF A GPU-BASED PARALLEL WIND FIELD MODULE

André L. S. Pinheiro[1], Roberto Shirru[2] and Cláudio M. N. A. Pereira[3]

[1] Programa de Engenharia Nuclear - COPPE
Universidade Federal do Rio de Janeiro
Centro de Tecnologia, Ilha do Fundão
21941-901 Rio de Janeiro, RJ
apinheiro99@gmail.com

[2] Programa de Engenharia Nuclear - COPPE
Universidade Federal do Rio de Janeiro
Centro de Tecnologia, Ilha do Fundão
21941-901 Rio de Janeiro, RJ
schirru@lmp.ufrj.br

[3] Instituto de Engenharia Nuclear - IEN
Comissão Nacional de Energia Nuclear
Rua Hélio de Almeida, 75, Ilha do Fundão
21941-906 Rio de Janeiro, RJ
cmnap@ien.gov.br

ABSTRACT

Atmospheric radionuclide dispersion systems (ARDS) are important tools to predict the impact of radioactive releases from Nuclear Power Plants and guide people evacuation from affected areas. Four modules comprise ARDS: Source Term, Wind Field, Plume Dispersion and Doses Calculations. The slowest is the Wind Field Module that was previously parallelized using the CUDA C language. The statement purpose of this work is to show the speedup gain with the optimization of the already parallel code of the GPU-based Wind Field module, based in WEST model (Extrapolated from Stability and Terrain). Due to the parallelization done in the wind field module, it was observed that some CUDA processors became idle, thus contributing to a reduction in speedup. It was proposed in this work a way of allocating these idle CUDA processors in order to increase the speedup. An acceleration of about 4 times can be seen in the comparative case study between the regular CUDA code and the optimized CUDA code. These results are quite motivating and point out that even after a parallelization of code, a parallel code optimization should be taken into account.

1. INTRODUCTION

When nuclear accidents occur with radionuclide leaks to the atmosphere, it is imperative to know where this radioactive plume is going. Accurate prediction of the direction of this plume is crucial to successfully guiding the teams responsible for protecting and evacuating people from possible affected areas. In order to predict the transport and diffusion of the radioactive material and its consequences for the environment, atmospheric radionuclide dispersion systems (ARDS) have been used [1]. ARDS are basically comprised by 4 main modules: Source Term, Wind Field, Plume Dispersion and Doses Calculations. In special, Wind Field and Plume Dispersion modules are the most computationally expensive.

This work focus in the Wind Field module. Wind Field models estimate wind conditions at each geographical point inside the region of interest. Here, the "Winds Extrapolated from Stability and Terrain" (WEST) model [2] is considered as the base for our development. In this model, the region of interest is divided into several volumes (cells) creating a large 3D block (as an array of 3x3 dimensions) where each cell has a component of the wind field like velocity and stability, as can be seen, in figure 1. Depending on how much the 3D grid is refined (decreasing the volume of the cell and increasing the number of cells), the computational overhead imposed may make their execution impracticable.

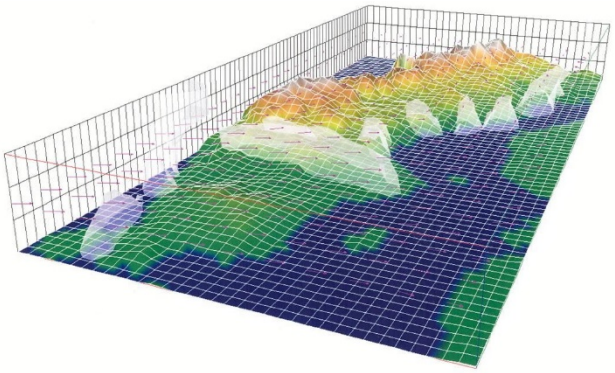

Figure 1: 3D grid example
Source: Adapted from http://www.smhi.se/en/research/research-departments/air-quality/match-transport-and-chemistry-model-1.6831, accessed 08/08/2017

Motivated by the fact that such computational overhead actually occurs in a realistic scenarios, it was developed a high performance parallel computational approach to Wind Field calculations, using WEST model, in order to allow the practical use of a high spatial resolutions. In the first approach made, a monolithic kernel structure was used where speedup of 40 times for interpolation of velocity and 10 times for stability interpolations was obtained, showing the viability of CUDA programming to solve the computational overload problem. This primary investigation is reported in Pinheiro et al., 2017 [3].

The second approach was to determine which part of the program consumed the longest processing time (in a sequential approach). As can be seen in table 1, the divergent minimization function consumes 92,23% of the total time of the program and this investigation is reported in Pinheiro, 2017 [4].

Table 1 – Relative processing time of functions in sequential Wind Field module.

Function name	Relative processing time
Stability Interpolation	0,62%
Vertical Extrapolation	0,00%*
Wind Field Initialization	0,16%
Land Addition	0,00%*
Speed Interpolation	5,95%
Speed Zero	0,02%
Transparency Calculation	0,27%
Divergence Minimization	92,23%

Divergence Removal	0,59%
Remaining West Function	0,01%* (Each one < 0,002%)

* Insignificant time ($<10^{-3}$%)

2. THE PARALLEL APPROACH WITH THE RELOCATION OF IDLE PROCESSORS

2.1. The Problem of Idle Processors

The third approach was to paralyze the code using the technique called Grid Stride Loop [5], but the parallel approach for Divergence Minimization function is not straightforward. The algorithm is iterative and presents strong sequential nature due to propagation of velocity correction signals (in order to minimize divergence) to the entire computational domain, which creates dependence between contiguous cells. Furthermore, it is by far the most time-consuming function of the program. By these reasons, the main focus here is to describe the most important issues and performance evaluation of the parallel version of Divergence Minimization.

In Pinheiro et al., 2017 [6], an intensive investigation was performed on the evaluation of the performance of the parallel version of the Divergence Minimization function and, at this moment, the 3D-Red-Black [5] approach was used as can be seen in figure 2.

```
__global__ void Divergence_Minimization_Kernel (parameters){
 ...
 // Calculate indexes in the matrix using the thread identification with 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)) || // RED and ODD sum
      ((color == 'B') && ((I + J + K) % 2 == 0)) { // or BLACK and EVEN sum
     IP1 = I + 1;
     ...
     // Calculate indexes in the big vector (3D matrix has been transformed into 1D)
     ...
     TOT = 1.0 / (TX[index_IP1] + TX[index_I]) / DX2 + (TY[index_JP1] + TY[index_J]) / DY2 +
                  (TZ[index_KP1] + TZ[index_K]) / DZ2;
     D = (VENTOU[index_IP1] - VENTOU[index_I]) / DX + (VENTOV[index_JP1] –
          VENTOV[index_J]) / DY + (VENTOW[index_KP1] - VENTOW[index_K]) / DZ;
     delta_Phi = 1.25 * TOT * D;
     U[index_I]   = VENTOU[index_I]   + delta_Phi * TX[index_I]   / DX;
     U[index_IP1] = VENTOU[index_IP1] - delta_Phi * TX[index_IP1] / DX;
     ...
    } // If   } // I  } // J } // K }
```

Figure 2: Divergence Minimization Kernel function

In the Divergence Minimization Kernel function, a big grid of threads is managed to process the entire partition (Black or Red partition). Note that the loops that appear in the algorithm refer to the grid-stride loop technique and the kernel is called twice: one for the Red pass and one for the Passage Black. As result of this approach, accelerations up to 24.91 times were obtained as can be seen in Pinheiro et al., 2017 [6].

In this version of the parallel algorithm, it can be noted that a big grid of threads is allocated to process the entire partition (Red or Black partition). However, due to the use of the 3D-Red-Black method, half of these threads become idle (In Red execution the threads containing the Black cells are idle and vice versa), which is not a problem for small computational domains that can be allocated in the threads available in the GPU, since all the cells are executed in a single cycle Parallel (even with half of them idle).

However, when the computational domain increases, there are not enough threads available in the GPU to execute all the cells of the computational domain in a single parallel cycle (each cell being executed by a thread) and due to this fact, the grid-stride Loop technique is used. In a simplified way, this technique creates within each thread an execution queue where the cells of the computational domain that could not be allocated due to the amount of threads available, are part of this queue and are executed in the later loops.

With the increase of the computational domain and the use of the grid-stride loop technique, half of the loops are idle, thus contributing to the increase of the time spent in executing the algorithm.

2.2. Relocation of Idle Processors

A Better memory management was been implemented so that the threads are allocated only with the Red cells, in the case of Red execution or only with the Black cells, in the case of Black execution, so that we no longer have idle threads. Figure 3 shows the implementation of the CUDA kernel with relocation of idle processors.

```
__global__ void Divergence_Minimization_Kernel (parameters){
 ...
 // Calculate indexes in the matrix using the thread identification with Grid-Stride Loops
 for (int K = blockIdx.z * blockDim.z + threadIdx.z; K < NZ; K += blockDim.z * gridDim.z){
    ...
   //Threads optimization
   KK = K * 2;
   if (Black)
    if ((I + J) % 2 != 0)
     KK--;
   else
    if ((I + J) % 2 == 0)
     KK--;
    IP1 = I + 1;
    ...
```

Figure 3: Divergence Minimization Kernel function with relocation of idle processors

2.2.1. Execution times and speedups

The objective of this section is only to quantify and analyze execution times and speedups obtained with the use GPU-based program with relocation of idle processors since the validation has already been done in Pinheiro et al., 2017 [6]. To accomplish that, a unique observed wind has been chosen (WId#2 [1]) and simulations with all refinement levels [6] and several number of iterations [6] were investigated. In the sequential code it was used an Intel-I7 2700K @3.50 GHz 3.90 GHz CPU and in the parallel code it was used GTX-680 GPU

Table 2 shows the comparative results between the execution times (in seconds) of the Wind Field module (average of 6 executions) for sequential implementations (CPU), first parallel implementation (GPU_1) and for parallel implementation with the relocation of idle processors (GPU_2) for the different levels of refinement and number of iterations. Computed times for GPU versions include the execution of GPU kernels, memory allocation in GPU and data transfer to GPU.

Table 2: Speedups and runtimes of sequential and parallel implementations

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

In this table, can be observed an increase of the speedup of this implementation (Speedup2) in relation to the one of the first implementation (Speedup1), thus demonstrating that the relocation of the idle processors had a positive effect on the performance of the algorithm.

Figure 4 shows the influence of the number of iterations at the execution time for all the computational domains and for the different implementations of the algorithm.

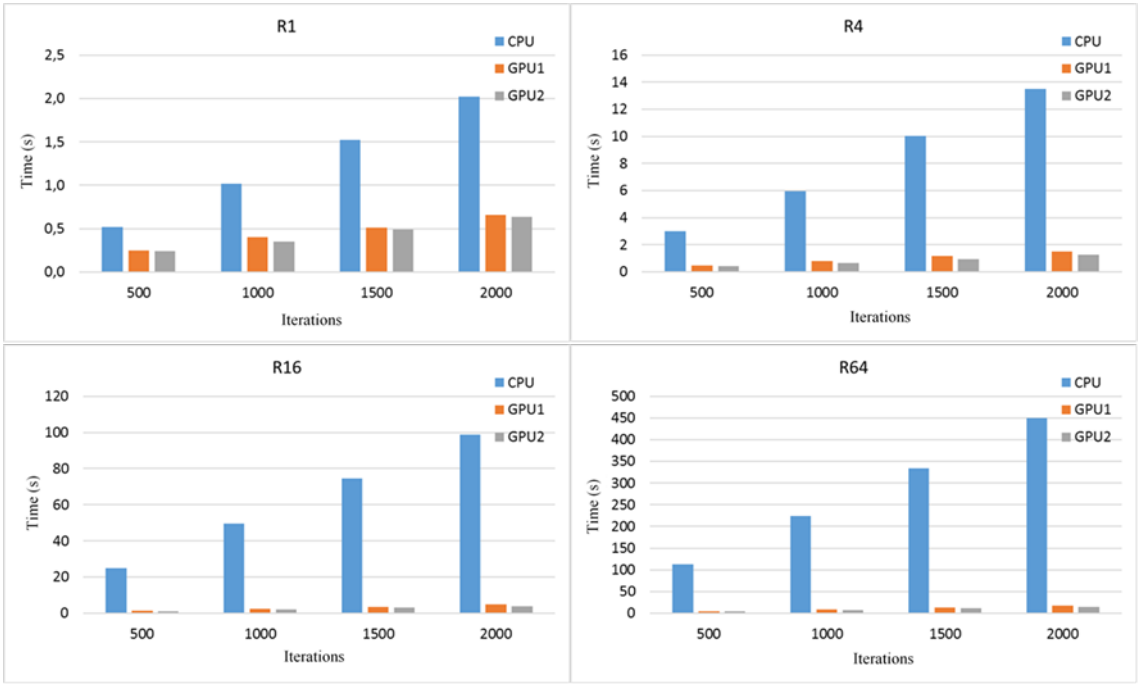

Figure 4: Execution time versus number of iterations for different computational domains:

In figure 5, can be observed the increase of the speedup after the optimization of the idle processors.

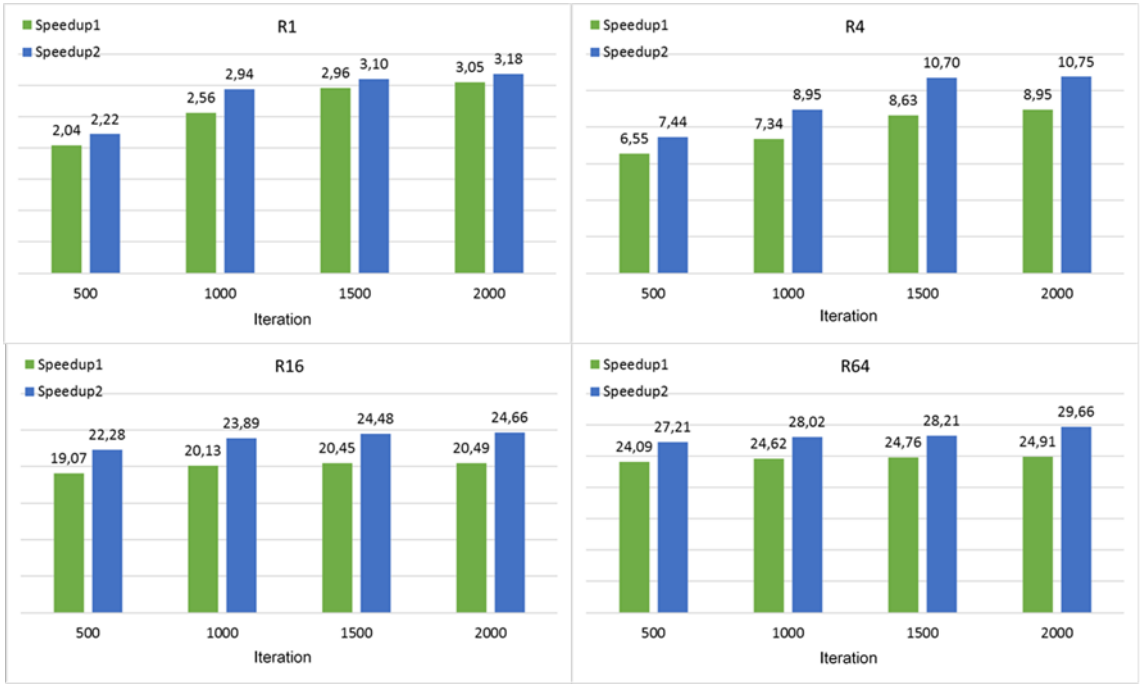

Figure 5: Speedup versus number of iterations for different computational domains:

In figure 6, can be observed increasing of the execution time as the computational domain is refined for the whole number of iterations investigated.

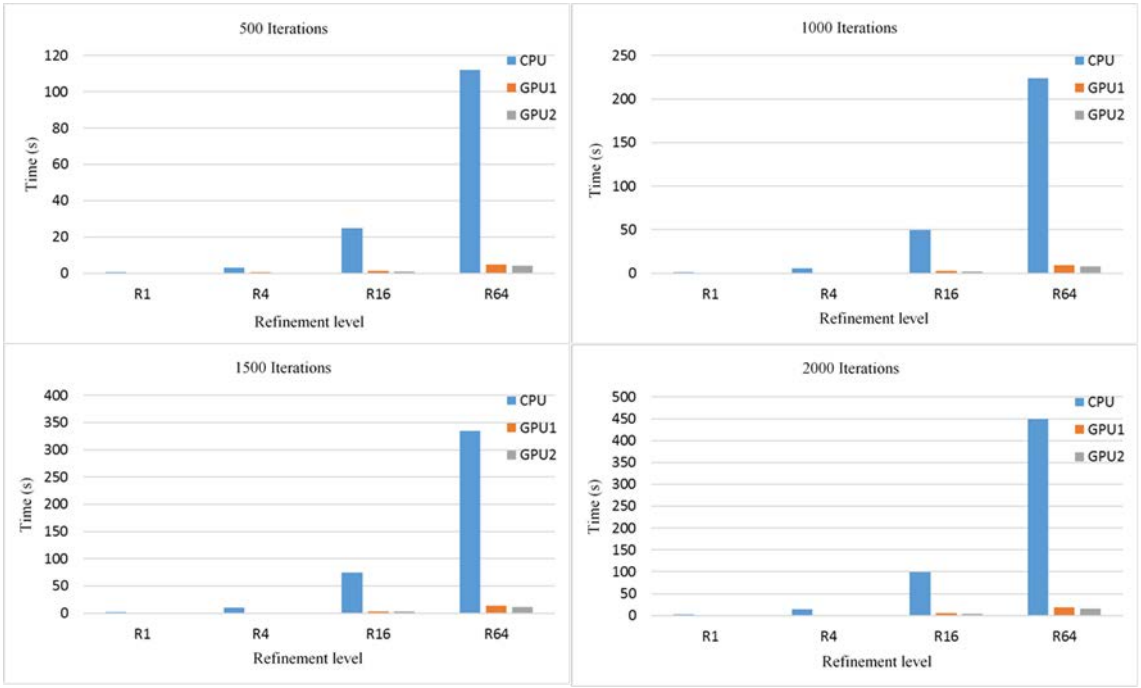

Figure 6: Execution time versus refinement level for different numbers of iterations

Figure 7 shows the comparative graphs of the speedups with the refinement of the computational domain.

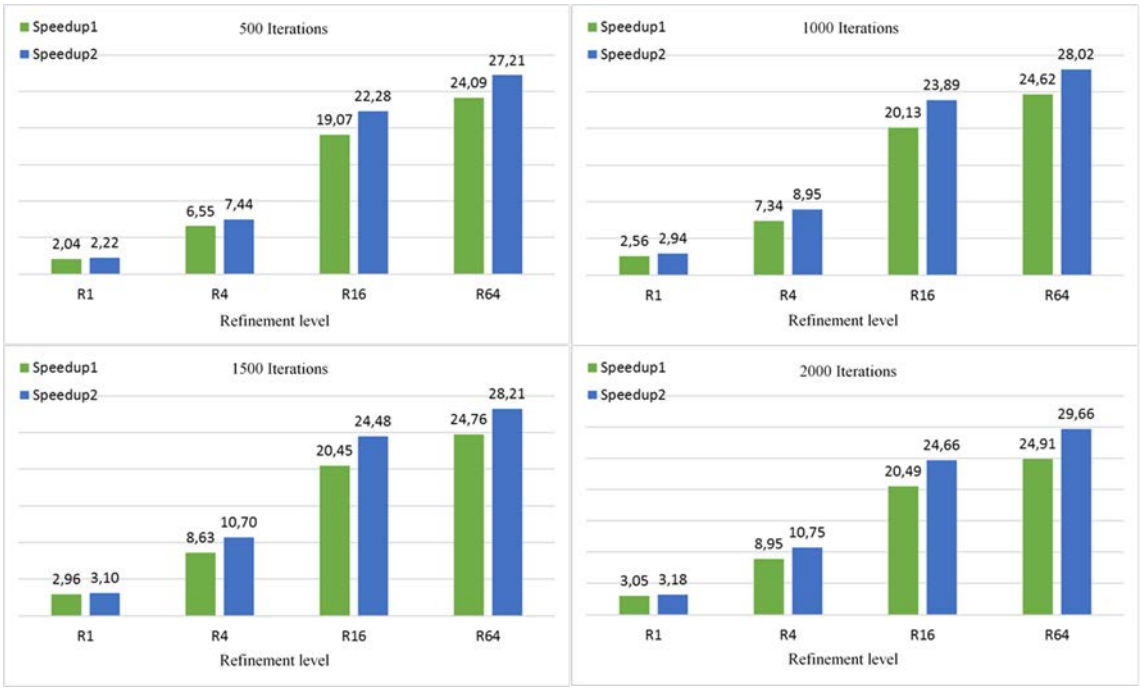

Figure 7: Speedups for different refinement levels for different numbers of iterations

3. CONCLUSIONS

As can be demonstrated, the execution time is approximately proportional to the number of iterations. Speedup is significantly reduced for simulations with fewer iterations and smaller computational domains, where the contribution of other functions in the total time is emphasized, increasing as the number of iterations and computational domain increases.

Comparing the results obtained in this work with the result of the previous work, we can observe as seen in table 3 a small increase in speedup. However, as seen in Figures 5 and 7, for larger computational domains or increase of iterations the increase of speedup becomes more significant in this way being of extreme importance that the idle threads must be reallocated.

Table 3: Speedup comparison

	Sequential Time	**Parallel Time**	**Speedup**
Previously work [6]	449,29	18,04	24,90
This work	449,29	15,15	29,66

As future works, it would be recommended that larger computational domains with larger numbers of iterations be studied to see the addition of speedup

REFERENCES

1. Park, S., Choe, A., Park, M. “Atmospheric Dispersion and Deposition of Radionuclides (^{137}Cs and ^{131}I) Released from the Fukushima Dai-ichi Nuclear Power Plant,” *Computational Water, Energy, and Environmental Engineering*, **2**, pp.61-68 (2013)
2. Fabrick, A., Sklarew, R., Wilson, J. “Point Source Model Evaluation and Development Study - The Grid Model IMPACT (Integrated Model for Plumes and Atmospherics in Complex Terrain),” *Technical report (Contract A5-058-87)*, California Energy.
3. Pinheiro A., Desterro F., Santos M., Pereira C., Schirru R., “GPU-Based Parallel Computation in Real-Time Modeling of Atmospheric Radionuclide Dispersion,” *Nunes I. (eds) Advances in Human Factors and System Interactions. Advances in Intelligent Systems and Computing*, Springer, Cham, Vol. 497, pp.323-333 (2017)
4. Pinheiro, A, “*Modelo Computacional Paralelo Baseado em GPU para Cálculo do Campo de Vento de um Sistema de Dispersão Atmosférica de Radionuclídeos,*” Universidade Federal do Rio de Janeiro – COPPE - Programa de Engenharia Nuclear, Rio de Janeiro, Brasil (2017)
5. “CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops,” https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/ (2013)
6. Pinheiro A., Desterro F., Santos M., Pereira C., Schirru R., “GPU-based implementation of a diagnostic wind field model used in real-time prediction of atmospheric dispersion of radionuclides,” *Progress in Nuclear Energy*, Vol 100, pp.146-163 (2017)