# Reducing runtime of WindNinja's wind fields using accelerators

Author: Carles Tena Medina

**Abstract–** Forest fire is a natural disaster that every year causes lots of losses. Human beings have always tried to fight them to prevent people from their effects. Currently, simulation tools and high performance computing arise as a powerful tandem to aid forest fire management. It is well known that wind is, by far, the most important factor in forest fire propagation and, furthermore, the wind speed and wind direction is modeled by the terrain topography. For that reason, it is mandatory to use wind field models that takes into account this features. WindNinja is a wind field model that provides for a particular terrain the corresponding wind field for a given meteorological wind. The obtained wind field can then be used in forest fire spread simulator (FARSITE in this work) to predict the evolution of a given wildland fire. Coupling a wind field model to a forest fire spread system provides better prediction results but the execution time of the whole systems increases. In this work, a method to speed up WindNinja execution using accelerators (GPUs) is described.

**Keywords–** WindNinja, wind field, forest fires, Conjugate Gradient with Preconditioner, CGP, GPU, CUDA, very sparse matrix

**Resum–** Els incendis son un desastre natural que cada any causen moltes pèrdues. Els humans sempre han intentat lluitar contra ells prevenint a la gent dels seus efectes. Actualment, les eines de simulació i la Computació d'Altes Prestacions són un poderós tàndem per ajudar a la gestió dels incendis forestals. Es sabut que el vent és, de lluny, el factor més important en la propagació d'incendis forestals i, per l'altre banda, la direcció i velocitat del vent es modelada per el terreny. Per aquesta raó, es obligatori l'ús de models de camp de vents que tinguin en compte aquestes característiques. WindNinja és un modelador de camps de vents que el calcula amb el corresponent terreny i un vent meteorològic donat. El camp de vents obtingut pot ser utilitzat per un simulador (FARSITE en aquest treball) per predir l'evolució d'un incendi forestal. L'acoblament del camp de vents al simulador proporciona millors resultats de predicció, però el temps de càlcul s'incrementa. En aquest treball es descriu un mètode per accelerar l'execució de WindNinja utilitzant acceleradors de còmput (GPUs).

**Paraules clau–** WindNinja, camp de vents, incendis, Gradient Conjugat amb Precondicionador, CGP, GPU, CUDA, matrius molt disperses

## 1 Introduction

Every year forest fires destroy many acres and, even in the worsts cases, lives. It's true that the best way to stop fires is their prevention, however, we cannot always stop them only with prevention and, therefore, predict their evolution is a knowledge that can aid to fight them. There are many simulation tools, which main goal is to predict the natural evolution of forest fires. Some of this simulators are FARSITE [4], FireStation [6], Wildfireanalyst [2] or CARDYN [7].

FARSITE is one of the forest fire spread simulators widely used in the forest fire scientific community. However, it was originally design to use wind values as homogeneous among the whole used terrain, as a consequence, the prediction accuracy was poor. In order to overcome such drawback, a wind field model has been coupled to FARSITE in order to consider an heterogeneous wind field and improve the forest fire prediction. The selected wind field modeler has been WindNinja [5]. WindNinja was originally developed by the same developer group as FARSITE, so it

E-mail de contacte: carles.tenamed@e-campus.uab.cat
Menció realitzada: Enginyeria de Computadors
Advisor: Anna Cortés Fité (CAOS)

was designed for being directly coupled to FARSITE.

This coupled forest fire spread prediction system, where the wind field has been considered, provides more accurate results than running FARSITE in a standalone scheme but, the total execution time of the system is increased due to the overhead included by the calculation of the wind field.

Since predicting forest fire evolution during an ongoing event requires to be able to deliver a prediction under strict deadlines, the final objective of this work is to execute the coupled system WindNinja + FARSITE with a hard real time restrictions. WindNinja has certain limitations that we must consider. On the one hand, it needs a big memory to save all the necessary data to be executed. Furthermore, WindNinja spends a lot of time to generate an output and, finally, WindNinja is not originally scalable.

WindNinja has 4 main parts: the equation creation, the array storage, the solver of the equations and, lastly, the wind field creation. Figure 1 shows these 4 components of WindNinja, which are described in more detail below.

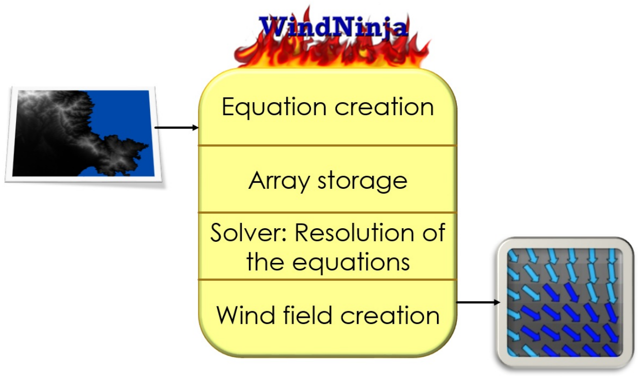

Figure 1: WindNinja parts

## 1.1 System of equations'creation

First of all, we have to calculate the equations that drive the system and, for doing this, WindNinja needs the elevation map of the terrain and the meteorological wind data (wind direction and wind speed) at the time that the fire is taking place.

The elevation map (DEM, Digital Elevation Map) consists of a raster file where the terrain is represented by cells of a certain size and, for each cell, the height of the particular point is stored. Furthermore, the relation of each point with its neighbours is needed and, for that purposes, the real distance between them is evaluated. Using this information, WindNinja builds a mesh where each node contains the information of each cell and in the nodes connections are the data of the relations.

Once the mesh is defined, the principles of mass conservation are applied to each node and each relation. The relation of between two node is the same independently of the order of the nodes, that is, one obtains the same value either if the mass conservations is evaluated from node $n_i$ to $n_j$ or in the backwards order. All obtained values will be later stored in a matrix. The main diagonal of this matrix includes the information of each node of the mesh, and the other elements of the matrix are the data of all the relations. As the relation between two nodes is the same if we calculate it onwards or backwards, the final matrix is a symmetric matrix and, because of this, we only need to store each element once instead of twice. In particular, the elements are stored in the superior part of the matrix what avoid redundancies and it requires less memory.

As a result, we have an upper triangular sparse matrix $A$. The original implementation of WindNinja stored this matrix in CSR [3] mode (later explained).

Besides, WindNinja needs the atmospheric wind speed and wind direction as input to generate a wind field. WindNinja applies also the principle of mass conservation in order to obtain a vector $b$ with the information at each mesh point related to the instantaneous homogeneous meteorological wind.

In the next section, we describe in more detail the storage scheme used by WindNinja (CSR) to keep the vector and matrix values in memory.

## 1.2 Matrix storage

As it has been previously mentioned, both the matrix $A$ and the vector $b$ are stored using the CSR scheme. To store a dens vector is easy but, keeping a sparse matrix in memory is more difficult. Our $A$ matrix is extremely sparse, the majority of the elements are zero and, for this reason, WindNinja uses a method who does not spend memory saving the zero elements. Because of this, WindNinja uses CSR method (Compressed Sparse Row) [3].

CSR uses three vectors to storage the matrix data. The firts vector, called $data$, saves the relevant data, rejecting the zero elements of the matrix. The other two vectors ($col\_index$ and $row\_index$) save the necessary information to position the data element in the matrix.

In particular, the $col\_index$ stores the information related to the columns. $col\_index$ is a vector with the same size that $data$ vector. Each position of the vector contains the column index positioned in $A$ of the $data$ element with the same index of the $col\_index$ vector.

On the other hand, the vector called $row\_index$ is a vector with the same size that rows have the $A$ matrix + 1. The first position of this vector always has a zero, the next positions stored the accumulative number of elements that have the row and the previous zone of $A$.

## 1.3 Solver: Resolution of the system of equations

Once we have obtained matrix $A$ and vector $b$,we have to calculate the following system of equation :

$$Ax = b$$

It seams that the best way to solve this systems consists of isolating vector $x$ like this:

$$x = A^{-1}b$$

This kind of resolution method has many problems. On the one hand, we cannot be sure that $A^{-1}$ is also a sparse matrix. On the other hand, the time spent to calculate the inversion is too much. In addition, the inversion matrix maybe could be too large to be stored in the available memory.

For that reason, the methods used to solve the system of equations are the Krylov methods [13]. In particular, within the Krylow's spaces, one has the iterative methods for positive defined sparse matrix, like the WindNinja ones. Specifically, the conjugate gradient with preconditioner (CGP) explained below.

The actual version of WindNinja with CGP is serial and it does not use all the potential that provides the actual parallel architectures such as multi-core and accelerators of computation. In order to take advantage of the computational improvements that this new computational architecture have, one has first to analyze how the CGP is implemented to, later, propose an alternative way of executing it on many-core architecture. For that reason, in the next section, we briefly describe how the CPG works.

### 1.3.1 Conjugate Gradient with Preconditioner

The purpose of including as a solver the CGP is to provide a good solution to the system of equations expressed by $Ax = b$, in a computational feasible time and faster that not using a Preconditioner. The algorithm that describes how to find the vector $x$ applying the CGP is shown in algorithm 1.

```
Starting from $x_0$
Calculate $g_0 = Ax_0 - b$, which is the difference between the initial value and the real value
Considering that M is the preconditioner evaluate
$q_0 = Mg_0$ and set the initial value of p as $p_0 = -q_0$
For k=1, ..., n:
    $\alpha_k = \frac{(g_k,q_k)}{(p_k,Ap_k)}$
    $x_{k+1} = x_k + \alpha_k p_k$
    $g_{k+1} = g_k + \alpha_k Ap_k$
    $q_{k+1} = Mg_{k+1}$
    $\beta_k = \frac{(g_{k+1},q_{k+1})}{(g_k,q_k)}$
    $p_{k+1} = q_{k+1} + \beta_k p_k$
```

Algorithm 1: Conjugate Gradient with Preconditioner (CGP)

Typically, the $x$ vector, that we want to found, lies at the intersection point of all the hyperplanes created by the quadratic form of each equation from the equations system.

To do this, one initializes $x$ and, at each iteration, one modifies it to be each time near to the real solution. These modifications are going to be explained by following the algorithm 1.

To obtain the best possible transformation of $x$ we have to calculate the vector $p$ who is going to modify it.

To do the modifications we need two scalars: $\alpha$ and $\beta$ that will modify the module of some vectors and, we also need three more vectors: $p$, $g$ and $q$.

First of all, we have to initialize the variables that we use in the algorithm to a concrete value to make shorter the convergence. After that, we start the modifications of $x$ that implies less iterations to converge, find the optimum result.

In order to find $p$, the first thing that we have to do is to calculate the vector $g$ using the previous version of $g$, the $A$ matrix, the previous version of $p$ and alpha.

When we have found $g$, we are ready to found $q$ that is an orthogonal vector perpendicular to $g$. Then, to obtain the result, we need the already calculated $g$ and the preconditioner.

Finally, we can find the $p$ vector using $q$ and the previous version of $p$ modified by $\beta$ that we also have to calculate before this.

At last, we are ready to modify $x$ using $p$, modified by $\alpha$, and the previous version of $x$.

This process is repeated for each iteration and we have to repeat it until the result becomes optimal

The optimum result of $x$ lies in the intersection of the maximum number of hyperplanes possibles. That depends on the reliability (error) of the result. Figure 2 shows in a graphical way, how this iterative process works.

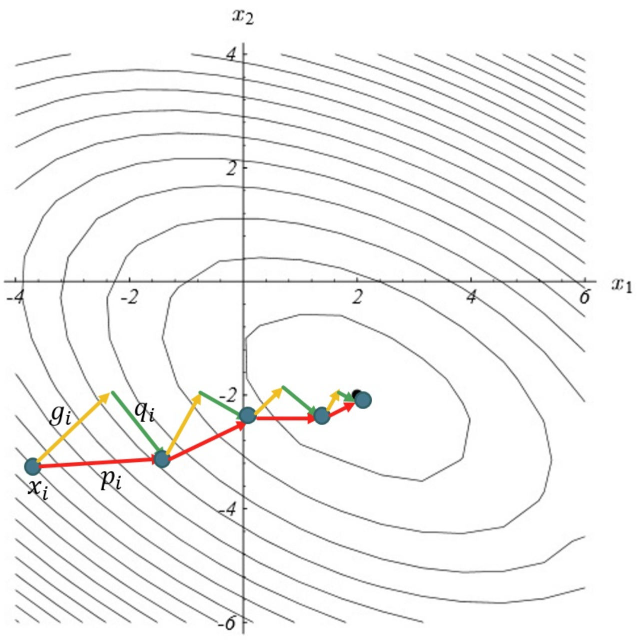

Figure 2: Conjugate Gradient with Preconditioner

## 1.4 Wind field creation

Finally, with the obtained results from the CGP (vector $x$), all obtained values related to one particular cell are regrouped. Consequently, one obtains a solution with the same number of elements that cells have the original map.

## 2 Objectives

As it has been previously introduced, the main objective of this study is to reduce the execution time of WindNinja in order to be able to generate a wind field evaluation at real time during an ongoing hazard. Otherwise, it will be impossible to deploy an operational system that could aid firefighter to take more accurate mitigation decisions. In particular, the part of WindNinja that takes more time is the solver of the equations system that spends, more or less, 80% of all execution time.

There are many strategies to speed up the solver but, we have focused to using accelerators, in particular, GPUs.

The main objective of this work, is to find an optimum strategy that takes the less time possible and it requires as few memory as possible to obtain the same results that WindNinja would deliver without any improvement.

## 3 State of the art

As it has been previously mentioned, the solver of WindNinja takes the major part of all the runtime (80%) and it is not paralleled at all. Furthermore, it has a very bad scalability when the size of the problem is increased.

There exist many strategies to reduce the execution time of WindNinja such as: partitioning the map Map Partitioning[11], reducing the resolution of the map, Map Resolution[9]. However, these solutions are not oriented to reduce the execution time of the solver itself, they are oriented to reduce the size of the system of equations and, therefore, it takes less time to run the solver.

There are others solutions, called domain decomposition, oriented to reduce the size of the mesh from which the system of equation is obtained. Some of them are oriented to use a better preconditioner such as: the Schwarz method[8] or the Schur method[10].

Other kind of strategies are oriented to parallelize the code of the solver using: OpenMP, MPI or news technologies like accelerator such as XeonPhi or, in our case, Graphics Processing Units (GPUs).

Another important issue to consider in this work, is the one related to the matrix storage strategy. Due to the particular structure of the matrix $A$, a new way to storage the matrix that improves a lot the runtime execution has been developed. It is possible because the matrix of WindNinja has a specifically pattern that makes possible the transformation of this sparse matrix to 14 dense vectors. It is a great innovation because reduce considerably the memory needed to storage the full matrix and improve the memory accesses to it. In this work, we have taken advantage of this new storage strategy and we have use the strategy of the vectorized matrix to make our solution[12].

There are other options, already in use, like use libraries like cuSparse[1] from NVIDIA, but none of them are efficient with a very sparse matrix like the one we are working with.

## 4 Methodology

In this study, first of all, WindNinja has been analyzed to understand how it works.

Secondly, the solver has been extracted from WindNinja and it has been deeply studied for a better comprehension.

Apart of this, we had analyzed our available computational platform and hardware and how they work. Then we had been doing smalls tests to get acquainted with the hardware doing little programs who work in the available GPU.

- Model: NVIDIA GeForce GTX TITAN.
- Memory: 6144 MB.
- Threads: 1024.

When all the concepts had been assimilated, the first thing that was tried was to transform all the solver using CUDA programming language. But, unfortunately, it didn't work.

Then, once we were more aware of the problem, we started doing small changes in the original code to transform every function to CUDA language, step by step.

When all the functions of the solver had been translated into the CUDA mode, we were able to unify different functions to a better and faster CUDA functions called kernels.

The problem comes out when a simple function needs more than one kernel. These kernels are not able to be unified and, it means that they have to be executed one by one, what implies that the second one have to wait until the first one have finished.

Because of this, we cannot unify all the functions in the same kernel, but, if it is discovered the way to do it, it will be a way to improve even more the execution time.

## 5 Results

Following the methodology described above, we have run the modify parallel CGP solver using matrix pattern storage for different matrix sizes. The number of elements of the matrix has been varied from 800.000 elements to 20 millions elements and the CGP has been executed in its serial version in a CPU, in CPU including the new storage pattern and executed in GPU with the same new storage pattern. The results in terms of execution time in seconds are listed in table the following table.

| N | MEM | CPU | CPU-Patron | GPU |
|---|---|---|---|---|
| 0.8 M | 353 MB | 285.70 | 204.32 | 5.17 |
| 1.8 M | 793 MB | 512.27 | 371.59 | 8.48 |
| 3.2 M | 1.38 GB | 1,038.86 | 756.21 | 16.58 |
| 5 M | 2.17 GB | 1,728.89 | 1,258.76 | 26.65 |
| 12.8 M | 5.56 GB | 5,075.35 | 3,784.40 | 78.09 |
| 20 M | 7.03 GB | 9.119.69 | 6,710.25 | killed |

Table 2: Runtime (seconds)

As it can be observed, the proposed strategy improves significantly reduce the execution time of the CGP solver keeping the times to a feasible time that could be considered to be included in a real time prediction system.

From the table, we can note that there is a case that no time has been provided. This particular case is the situation where the obtained matrix required more than 6 GB what is the maximum memory capacity of the used GPU. In this case, the execution is killed and no results are reported.

So, despite the case where the matrix is too big for the current version of the proposed implementation on GPUs, the new approach provides greats improvements in terms of time savings enabling the capability of performing wind field evaluations at real time.

## 6 Conclusions

There exists many libraries that offer solutions to run CGP in parallel such as cuSparse, PETSc, ... but the kind of matrix generated in our problem is too disperse and it has a bad scalability when applying these solutions. We should provide a comparative study in terms of time to these alternatives but they have not been implemented yet. However, our expectations are that they cannot improve our results using pure CUDA language programming.

To sum up, we have achieved really great results in terms of execution time reductions when using GPUs, although we think that they can be improved if additional enhancements are included the described proposal.

One way to improve the solution could be to study how a reduction function can be executed in an unique kernel, and when it becomes possible, unify all the kernels in only one and take advantage of the different kinds of memory of the GPUs to make it faster.

Apart of this, we think that nowadays the options to develop code to run in the GPUs is a bit rudimentary, like the other ones in the past, and it is a problem to develop code faster and easily. But with hard work you can obtain really great results. We think that in the future will appear new options to do it efficiently and with better results.

## REFERENCES

[1] cusparse. http://docs.nvidia.com/cuda/cusparse: Last visited on 17/06/2015.

[2] Tomàs Artés, Andrés Cencerrado, Ana Cortés, Tomàs Margalef, Darío Rodríguez-Aseretto, Thomas Petroliagkis, and Jesús San-Miguel-Ayanz. Towards a Dynamic Data Driven Wildfire Behavior Prediction System at European Level. *Procedia Computer Science*, 29:1216–1226, 2014.

[3] Aiyoub Farzaneh, Hossein Kheiri, and Mehdi Abbaspour Shahmersi. an Efficient Storage Format for Large Sparse Matrices. *Communications, Faculty Of Science, University of Ankara Series A1Mathematics and Statistics*, 0:001–010, 2009.

[4] Mark A Finney. *Farsite fire area simulator - model development and evaluation*. Ogden, UT [Department of Agriculture, Forest Service, Rocky Mountain Research Station, 1998.

[5] Jason Forthofer, Kyle Shannon, and Bret Butler. Simulating diurnally driven slope winds with WindNinja. *Proceedings of 8th Symposium on Fire and Forest Meteorological Society*, page 13, 2009.

[6] a. M G Lopes, M. G. Cruz, and D. X. Viegas. Firestation - An integrated software system for the numerical simulation of fire spread on complex topography. *Environmental Modelling and Software*, 17:269–285, 2002.

[7] J. Martinez Millan, S. Vignote, J. Martos, and D. Caballero. Cardin, un sistema para la simulación de la propagación de incendios forestales. *Forest Systems*, 0(1), 2008.

[8] Gemma Sanjuan and Ana Cortés. Applying Domain Decomposition Schwarz Method to Accelerate Wind Field Calculation. *International Conference on High Performance Computing Simulation 2015. CORE B.*

[9] Gemma Sanjuan, Ana Cortés, and Tomàs Margalef. Effect of map resolution on wind field accuracy Prediction. *International Conference on Fire Behaviour research 2015.*

[10] Gemma Sanjuan, Tomàs Margalef, and Ana Cortés. Applying Domain Decomposition to Wind Field Calculation Parallel Computing Special Issue on Parallel Matrix Algorithms and Applications. *Parallel Computing Systems Applications.*

[11] Gemma Sanjuan, Tomàs Margalef, and Ana Cortés. Adapting Map Resolution to Accomplish Execution Time Constraints in Wind Field Calculation. *Procedia Computer Science*, 51:2749–2753, 2015.

[12] Gemma Sanjuan, Carles Tena, and Ana Cortés. Applying vectorization of diagonal sparse matrix to accelerate wind field calculation. 2015.

[13] Ha Van Der Vorst. *Iterative Krylov methods for large linear systems*. 2003.