Environmental Modelling & Software 38 (2012) 167–177

Contents lists available at SciVerse ScienceDirect

Environmental Modelling & Software

journal homepage: www.elsevier.com/locate/envsoft

Simulation of surface fire fronts using *fireLib* and GPUs

F.A. Sousa, R.J.N. dos Reis, J.C.F. Pereira[*]

Instituto Superior Técnico, Technical University of Lisbon, Mech. Eng. Dept., Lasef, Av. Rovisco Pais 1, 1049-001 Lisboa, Portugal

A R T I C L E I N F O

Article history:
Received 29 March 2011
Received in revised form
25 May 2012
Accepted 6 June 2012
Available online 6 July 2012

Keywords:
GPU
Cellular automata
Surface fire propagation
Rothermel
BEHAVE

A B S T R A C T

This paper reports the porting of raster type fire propagation models to graphical processing unit (GPU) architectures. Two known fire growth model algorithms were ported and a new one developed. All models are based in cellular automata, programmed in the CUDA framework developed by NVIDIA and their results compared against serial execution. The novel algorithm targeted for GPU architectures achieved a speedup (SP) of over 200× against serial CPU execution. These results allow ultra *faster-than-real-time* capabilities that may prove useful toward fire fighting methodologies.

Software availability

Name of software: CudaFGM
Developers: LASEF-IST
Contact address: LASEF/DEM/IST, Pav Mex I, Av Rovisco Pais, 1049-001 Lisboa Portugal
Email: jcfpereira@ist.utl.pt
Availability and Online Documentation: All the source code is available under request to the authors, being licensed under GPL V2.
Year first available: 2012
Hardware required: CUDA devices with compute capability 1.1 and above.
Software required: CUDA SDK
Programming language: C/CUDA
Program size: 8 MB.

1. Introduction

Forest fires create complex scenarios because of the interaction between the fire front, airflow and terrain. To better allocate resources, which in many occasions are scarce, and decide what actions to take, the decision actors need to predict the evolution of the fire front. Unfortunately, due to the mentioned complexity of the forest fire, computational tools that help predict the fire behaviour are still far from perfect. Such reality introduces the need to be able to run different scenarios, gauging these tools sensibility to different stochastic input parameters and still be fast enough to be of use during the fire fight, *i.e.*, within a stochastic approach framework. Current codes are able to deliver faster-than-real time results for deterministic, single case scenarios. Stochastic approaches demand, however, the ability to run hundreds or thousands of cases and, consequently, a two order increase in speed-up on these codes may be required. This is achieved by exploiting the fine-grain parallelism in forest fire algorithms so they are able to profit from the emerging technology of the graphical processing unit (GPU) for general computation.

GPU hardware has shown to be able to sustain very high speedups, accelerating time-to-result, in an increasing number of applications. They are also almost ubiquitous in the computing platforms sold in the consumer market, becoming a very interesting choice for a cheap, high computing power machine for use with computing aid decision tools for forest fire fight.

Members of the scientific community have used GPUs as soon as their capability to explore fine-grained parallelism was perceived. The introduction of better and easier GPU programming frameworks by NVIDIA and other vendors has fostered the exploitation of this type of parallelism, resulting in reports of impressive speedups in many areas. Matrix vector operations, critical in many applications, already have versions with two orders of magnitude speedups (see *e.g.* Bell and Garland, 2008; Wiggers et al., 2007; Bolz et al., 2003). Computational Fluid Dynamics (CFD) engineering

[*] Corresponding author.
E-mail address: jcfpereira@ist.utl.pt (J.C.F. Pereira).

1364-8152/$ – see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.envsoft.2012.06.006

tools have been implemented in GPU architectures. The double precision compressible Navier–Stokes solver with the Boussinesq approximation is reported by Cohen and Molemaker (2009) to be 8 times faster than the calculations obtained in the 8 core Intel Xeon at 2.5 GHz. Incompressible Navier–Stokes solutions of flows in complex geometries also displayed one order magnitude speedup (see Elsen et al., 2008) and higher speedups have been obtained with Lattice Boltzmann and particle methods (see *e.g.* Tolke and Krafczyk, 2008; Habich, 2008). Extensions to GPU clusters are also under way (see *e.g.* Phillips et al., 2009; Thibault and Senocak, 2009). Of the several programming frameworks alternatives for this type of hardware, CUDA, from NVIDIA (see Kirk and Hwu, 2009), is the most mature, consisting in a series of C language extensions.

To exploit the capabilities promised by the GPU, the fire spread algorithms must be designed with fine grain parallelism in mind. Two main approaches have been developed for prediction of forest fire spread: vector and cellular automata. The last becomes a natural choice for fine grain parallelism exploration as the former implies several branching options during the algorithms execution, making them less suitable for the GPU architecture.

Forest fire modelling has been developed under different approaches (see, *e.g.* the review by Pastor et al., 2003) and, according to Sullivan (2009a,b), three kinds of modelling strategies can be defined: *empirical*, in which no modelling is required; *semi-empirical* (or *semi-physical*), based on Rothermel (1972), where no distinction is made between different heat transfer modes; *physical models*, which differentiate between the various kinds of heat transfer mechanisms: *e.g.*, multiphase models, embodying Navier–Stokes equations with turbulence models, combustion in porous media, etc... (see *e.g.* Larini et al., 1998; Mell et al., 2007; Morvan and Dupuy, 2004; Costa et al., 1995; Zhou and Pereira, 2000).

Of the different approaches mentioned, semi-empirical models are present in most of the forest fire computer-aided decision tools currently in use and can be traced back to the pioneer work of Rothermel in the 1970s. He introduced a semi-empirical mathematical model to characterize quasi-steady unidimensional surface fire propagation, setting the rate of spread as a function of forest substract (the fuel), wind velocity, direction and slope (Rothermel, 1972). Later, the Rothermel model was extended by such models as BEHAVE, which added, for instance, fire spotting (hot embers projected ahead of fire front) and the implementation of the thirteen NFFL (Northern Forest Fire Laboratory) fuel models (see the BEHAVE model for details: Andrews, 1986; Andrews and Chase, 1989; Burgan and Rothermel, 1984). Detailed reviews of these models can be found in, for instance, Viegas (1998), Pastor et al. (2003) and Papadopoulos and Pavlidou (2011).

BEHAVE has been incorporated into several commercial fire modelling tools as Flamap (Finney, 2006), Firestation (Lopes et al., 2002) and Farsite (Finney, 1998). BEHAVE is also made available through the highly optimised library *fireLib*. Because *fireLib* is made available in source code it is an excellent tool for the development of fire spread algorithms or testing their deployment in new computational architectures.

This work is mainly concerned with the GPU implementation of semi-physical models for forest fire growth to speedup the calculations. Former approaches to exploit parallelism in forest fire computational tools have explored data or functional parallelism (see *e.g.* McDonough and Garzón, 1999; Jorba et al., 2001; Innocenti et al., 2005). Functional parallelism, where several tasks can be simultaneously executed, with or without data exchange, is convenient for deployment in GRID computing environments (see *e.g.* Rodriguez et al., 2010). Innocenti et al. (2009) exploited fine-grain parallelism for fire prediction using OpenMP in multi-core architectures but reporting limits on their approach regarding achieved speedup. To the authors knowledge, their work is the first that explores fine-grained parallelism for forest fire prediction in the GPU.

The objective of this work is to investigate the computational performance of fire spread models using GPUs and comparing their speedup against sequential CPU processing. Several fire propagation models were ported to the GPU using CUDA and verified against published results. These models have shown a very significant performance increase after being expressed by algorithms specially developed for the GPU. The algorithms developed are suitable for implementation in other GPU exploiting frameworks (*i.e.* OpenCL).

The next section discusses the algorithm options to port *fireLib* and three fire growth models into the CUDA framework. After, several results are presented, ranging from fires in simple flat terrain uniform vegetation to fires in a realistic complex terrain. Finally, summary conclusions are withdrawn regarding the speed gains due to the parallelisation strategy employed.

2. Fire growth models

2.1. Fire modelling approach

Roughly speaking, two functional parts are coupled together to achieve a fire prediction algorithm. The first part computes, using data about the terrain topography, fuel, wind, etc, the fire rate of spread (ROS) from the already ignited/burning points in the terrain. This is achieved, for instance, by the algorithms of the BEHAVE model available through the *fireLib* software library. The second part uses the calculated ROS to propagate the fire in the terrain (*i.e.* it is the fire growth model (FGM)). In this respect two main strategies are generally used: vector and raster.

Vector based fire growth models, like the one in Farsite (Finney, 1998; Anderson et al., 1982), discretize the fire front in several points which are then iteratively propagated. At each time step, and for each point, the ROS is computed and with this information the new point position obtained.

Raster models, on the other hand, discretize the terrain itself into cells and use the calculated ROS to propagate the fire in a contagion fashion. The end result is a terrain map were for each cell an ignition time is presented (*i.e.*, the time when the cell has started to burn).

For the purpose of the work here presented (deployment in GPU architectures) a raster approach was chosen as it was considered more amenable to the parallel paradigm imposed by the GPU.

The ROS computation is usually done inside the FGM cycle using the following algorithm:

i Obtain the rate of spread disregarding the wind and terrain influence (ROS_0);
ii Obtain the propagation ellipse from the ROS_0, correcting it to account for wind speed and direction and terrain slope and aspect. The ellipse is characterized by its maximum direction of propagation (α), maximum rate of spread (ROS_{max}) and eccentricity (E).
iii Obtain the ROS for each neighbour. The ROS is obtained according to:

$$ROS = ROS_{max}\frac{1-E}{1-E\cos(\theta)} \tag{1}$$

where θ is the angle between α and the neighbour direction.

As mentioned, the input parameters feed to the models are organised in maps where each element has a direct correspondence with each cell of the discretized terrain used by the FGM algorithm. Each input variable carries its own map (for instance, wind direction, wind speed, slope, aspect, fuel type and fuel model). The output data is an ignition map.

Three raster fire growth models were implemented in detriment of others, like the vector propagation techniques used in FARSITE (see Anderson et al., 1982). Raster fire modelling has a much more straightforward approach compared with vector propagation.

2.2. Minimal Time (MT)

The Minimal Time FGM can be found, for instance, in BFOLDS (Perera et al., 2008) and FireStation (Lopes et al., 2002). It is a contagious algorithm which propagates the fire from each cell to its neighbours according to the stencil in Fig. 1. In this stencil a different number of neighbours can be considered and a 16 cell scheme was used in the work here presented.

The time of ignition of each neighbour is calculated from its centroid distance to the cell centroid (L) and the ROS in that direction ($t_{\mathrm{ig}} = L/\mathrm{ROS}$).

The algorithm progression time step (Δt) is determined from the minimum time needed to ignite an unburnt cell in the map at the next iteration step. Due to this Δt is likely to change during the iteration process. The evolution set of rules leading the system can be summed up as:

- For a given iteration step n, with simulation time t^n, the cell is burning if $t_{\mathrm{ig}} = t^n$. If so the neighbours ignition times t_{ig_V} are computed using $t_{\mathrm{ig}_V} = t^n_{\mathrm{ig}} + L/\mathrm{ROS}$, where V are the vicinity cells forming the neighbourhood stencil (see Fig. 1). For each the computed value is compared with the previous stored ignition time and replaces it if it is lower. This is done to atone to the fire arrival times in multiple fire ignition situations or concave fire fronts.
- if $t_{\mathrm{ig}} < t^n$, the cell is considered burned and is ignored;
- if $t_{\mathrm{ig}} > t^n$, the cell is unburnt and ignored.

t^{n+1} corresponds to $\min(t_{\mathrm{ig}} > t^n)$ of all cells in the end of iteration step n.

2.3. Fixed time step (FT) FGM

This model departs from the previous by using a fixed time step (Δt) and an algebraic rule to assess the cell state change (see Karafyllidis and Thanailakis, 1997; Encinas et al., 2007). Because the algebraic rule is dependent on the number of neighbours and no reference to higher order stencils were found, only an eight cell stencil was considered (see Fig. 2).

Each cell has a state S expressing the ratio of burned area to total cell area: $S = A_b/A_t$, with $S \in [0,1]$. Propagation of the fire is then dependent on S through:

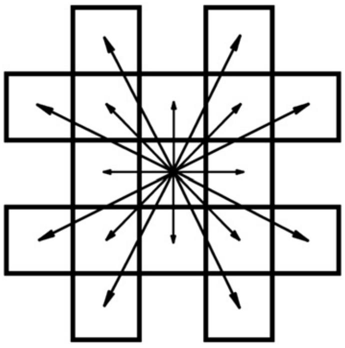

Fig. 1. Neighbourhood and propagation paths for the Minimal Time FGM.

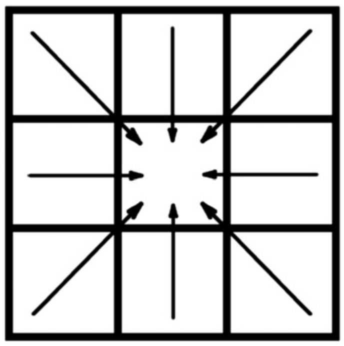

Fig. 2. Neighbourhood and propagation paths for the Fixed Time Step FGM.

- $S = 0$, cell is unburned;
- $S = 1$, cell had burn completely. This cell will only stop to be taken into account when all its neighbours are also with $S = 1$.
- The cell state S is a function of the neighbours state S^n_V on the last iteration step n, plus its own state at S^n, as given in the transition rule:

$$S^{n+1} = S^n + \sum_V^{\text{adj.}} \frac{\mathrm{ROS}_V}{L_V/\Delta t} S'^n_V + \sum_V^{\text{diag.}} \frac{\pi \mathrm{ROS}_V^2}{4(L_V/\Delta t)^2} S'^n_V \qquad (2)$$

where “adj.” and “diag.” refer to the adjacent and diagonal neighbours (Fig. 2), and $S'^n_V = 0$ if $S^n_V < 1$ and $S'^n_V = 1$ otherwise. The different treatment given to diagonal and adjacent cells is due to the way the fire propagates: with a straight front between adjacent cells and with a circular front between diagonal cells, centred in the vertex of the burned cell.

2.4. Iterative Minimal Time (IMT)

Due to the different programming computing paradigm imposed by the GPU architecture, the algorithm implementation strategy of the previous models was found lacking in performance. Using the knowledge acquired in porting these algorithms to the GPU a new approach was deemed necessary.

A new model was developed departing from the Minimal Time fire growth model and keeping its physical prediction characteristics. This new model, called Iterative Minimal Time (IMT), is a numerical iterative procedure characterized by data locality, *i.e.* the execution of the propagation rules is fully independent for each cell. The major difference from the MT model is that the propagation is now being made from neighbour to local cell and, through a discrete numerical iterative scheme, decoupled from the discrete time progression. In other words, each iteration of the initial MT model corresponds to a real physical solution, while each iteration of the new IMT model is just a numerical solution, without any physical meaning attached. The physical solution of interest is obtained when the method converges.

The ignition time map converges iteratively, starting from an approximate solution or initial guess. Each new iteration $k+1$ uses the k ignition times to compute the new ignition time for each cell (t^{k+1}_{ig}) in the domain. Using Fig. 3 stencil, t^{k+1}_{ig} is computed as the minimal propagation periods from the neighbourhood cells, according to

$$t^{k+1}_{\mathrm{ig}} = \min\left[t^k_V + \frac{L_V}{\mathrm{ROS}_V}\right] \qquad (3)$$

where t^k_V are the old ignition times of the neighbour cells V, L_V is the centre point distance between centre and neighbour cells and ROS_V is the rate of spread in the neighbour-local direction. A solution is

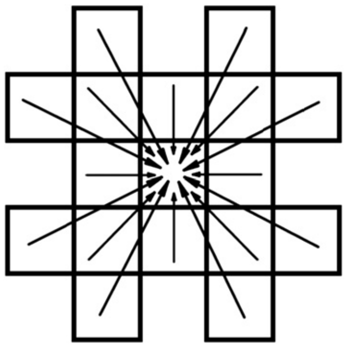

Fig. 3. Neighbourhood and propagation paths for the Iterative Minimal Time FGM.

found when the difference between the two fields, k and $k+1$ is zero.

3. The GPU architecture

The GPU was developed as a hardware device dedicated to process large graphic data sets, freeing the CPU from this task. Because of this, the GPU and CPU architectures developed along different lines. The CPU, in one hand, must handle the quasi-simultaneously processing of different software programs and also manage a large range of hardware devices, connected through an ensemble of different bandwidth buses. An important share of the CPU electronics is thus dedicated to control and management logic. To cope with all the different instruction and data streams simultaneously present, other substantial amount of chip space is dedicated to an hierarchy of caches, to hide latency from the user.

The GPU, on the other hand, is setup for the specialized graphic processing task, being geared towards throughput instead off calculating speed *i.e.*, the target metric is data processed per time unit instead of faster instruction execution. The result is that the majority of its electronics is dedicated to arithmetic operations in the form of a large number of low frequency (relatively to the CPU) *Streaming Multiprocessors* (SM). This characteristic also sidesteps the problems found by CPU designers in taking advantage of Moore's Law (Moore, 1965): instead of attempting ever higher frequencies and facing heat problems, GPUs increase the number of arithmetic processors in each device and implement faster and wider memory buses, an area still expected to grow with Moore's Law. This last aspect is also fundamental to the GPU design, the presence of large, dedicated global memories, able to feed the streaming multiprocessors through a highspeed, high bandwidth bus (for instance, the NVIDIA GT200 chip supports about 150 GB/s while single core CPUs are not expected to overcome 50 GB/s (Kirk and Hwu, 2009)).

The exploitation of this hardware is made possible through the execution of a very large number of threads simultaneously (in the order of thousands). Each thread processes a chunk of data concurrently to other threads, in a parallel fashion. The GPU is able to run hundreds of these threads at almost no cost because the thread management logic is implemented at the hardware level. In the CPU, on the other hand, threads are managed by the operating system, being expensive to create and destroy.

It is clear from the exposed that not all problems are amenable to GPU exploitation. Because of that, the best suited problems for the GPU are those displaying a high degree of operation parallelism, allowing simultaneous, independent processing of large amounts of data, *i.e.*, Single Instruction Multiple Data (SIMD) type of algorithms.

At the hardware level the several Streaming Multiprocessors present on the GPU device access and operate different memory levels, like the CPU, but with higher bandwidth buses. These memories are accessed and used with different roles: a *Global memory* RAM space with direct communication with the *host* (CPU) RAM. This memory is accessible by all threads and has the highest latency access times present in the device. The SMs also have access to *Register* memory. This is the fastest available memory but only accessible to each thread. It has a small size, demanding careful management.

Because of its specialized nature, GPU programming introduces or renders more acute several problems to the programmer, namely, memory management issues, load balance of the executing threads, data races, data transfer costs, among others. For a review and comment on these see Kirk and Hwu (2009).

A GPU/CPU program workflow follows the general idea of off-loading operations and data to the GPU (*device*) from the CPU. Operations are executed through *kernel* functions, that can operate in the CPU, CPU and GPU or just GPU. A generic example of instruction flow in the GPU is given in Fig. 4.

CUDA, or Compute Unified Device Architecture, was developed by NVIDIA as an ANSI C extension to facilitate the programming of this manufacturer's GPUs (NVIDIA, 2009). It was the programming framework used to develop the work present albeit conclusions and results are extensible to other GPU programming approaches.

4. Mapping the algorithm to the GPU

All implemented fire growth models mentioned in this work start by building a fuel model catalogue from pre-defined (see, for *e.g.* Andrews, 1986; Scott and Burgan, 2005) or user defined input. An iterative loop follows where, at each iteration, the propagation rules of one of the algorithms in Section 2 are applied to all cells. The main loop stop condition is dependent on the fire growth model chosen: it can be a limiting the number of iterations, a time limit, convergence criteria or burning of all cells in the map.

Fig. 5, represents the algorithm flowchart and roles of the host and the device (GPU). The CPU controls the catalogue setup, all data

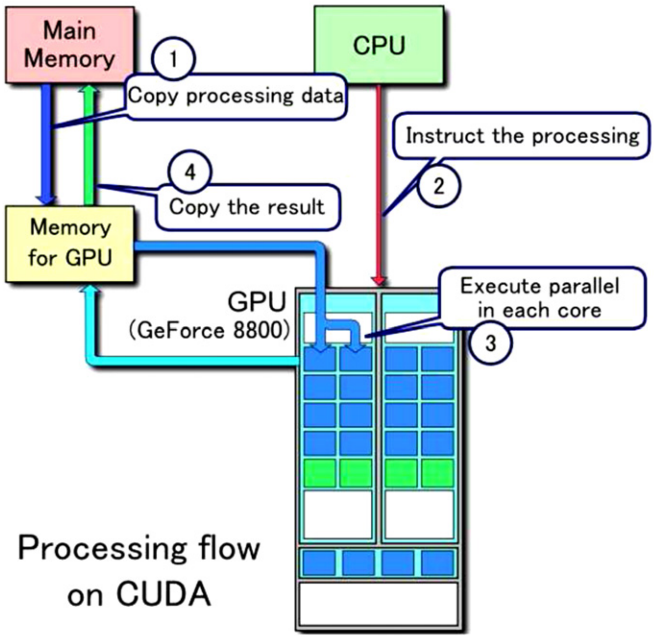

Fig. 4. GPU data flow example (Tomaka, under ⓒⓒ).

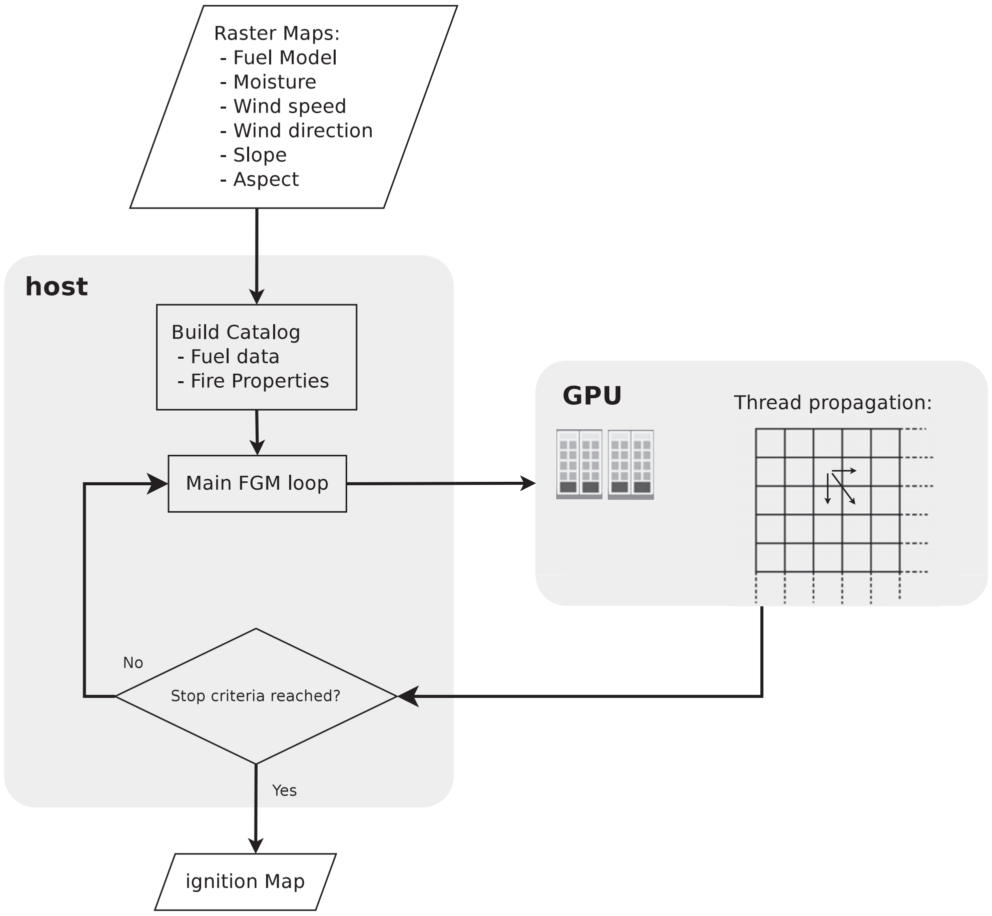

Fig. 5. FGM flowchart highlighting the interaction between the host and the GPU. The GPU block (to the right) depicts a thread conducting the fire propagation to unburned neighbour cells.

input/output and main execution loop. In the GPU implementation, the spatial cell propagation part, inside the main loop, is offloaded to the device and run inside a kernel.

A thread is assigned to each cell. Threads can be processed independently (*i.e.*, in any order), having a unique identification assigned by the CUDA framework. Here lies one of the main differences from serial programming: the sequential loop over all cells is replaced by an implicit loop, where all elements can be calculated at the same time, restricted only by the device memory and processing availability.

The different models implementations are described bellow.

4.1. Minimal Time kernel

The minimal time model described in Section 2.2 highlights the problems found of porting on algorithm between two different computational architecture paradigms.

The stop criteria for this model is the full burn of the map area or a user defined time limit. This is controlled by the main outer loop, running on the CPU. During the fire propagation phase, offloaded to the GPU, a global search loop identifies all burning cells and applies the rules described in Subsection 2.1. The ignition time in the stencil cells is then computed from the calculated ROS. In the end, the minimum ignition time of all cells is returned to the CPU obtaining the next time step, *i.e.*, $t^{n+1} = t^n + \min(\Delta t_i)$. If the stop condition is reached the full ignition map is transferred to the CPU for output.

Several limitations where found with the deployment of this model into the GPU. The first such limitation arose from the number of instructions needed to encode the ROS propagation rules for this model. As the memory available in each of the device stream processors must be shared between instructions and local data, the kernel size is limited. Part of the algorithm was thus displaced to the outside of the fire growth model in the following manner: steps i and ii in Subsection 2.1 are run in the GPU in a dedicated kernel before the main loop, obtaining the α, ROS_0, ROS_{max}, and E and storing these values in dedicated arrays in the device main memory. These arrays are used by a second kernel to compute step iii.

A second limitation is incurred when updating the cells as each thread can potentially alter the ignition time values of any of its stencil cells. To avoid data corruption due to data races, *i.e.*, write to the same memory location by a thread while other thread is reading it, atomic operations must be enforced (done here using the CUDA function *atomicExch*). This safe-guard measure represents a performance penalty because memory access becomes serialized (*i.e.*, cannot be performed in parallel).

For comparison, pseudo-code implementations of this algorithm for CPU and CPU/GPU are given, respectively, in Figs. 6 and 7.

4.2. Fixed Time Step kernel

The fixed time step algorithm eliminates the need for atomic operations: each threads writes in its own dedicated memory space due to the inward “fire spread” direction of the propagation stencil (see Fig. 2).

Like in the previous algorithm, the main outer loop operates over time but using a user defined, fixed time step, *i.e.*,

```
timeNext = 0;
while ( timeNext < INF ){
  timeNow  = timeNext ;
  timeNext = INF;

  // loop all CELLS in the domain
  for (cell = 0; cell < CELLS; cell ++){
    if (timeNext > ignMap[cell] and ignMap[cell]  > timeNow)

    //Update timeNext
    timeNext = ignMap[cell];

    //Find burning cells
    else if (ignMap[cell] == timeNow){

      //Propagate from burning cells
      for (n = 0 ; n < 16; n++ ){
        ncell = //..compute neighbour cell index

        //If neighbour is unburned
        if ( ignMap[ncell] > timeNow ){

          //Compute ignition time
          ROS = //...Compute ROS in the local-neighbour direction
                // with  (equation 1)
          ignTime = timeNow + L_n / ROS;

          //Update neighbour's ignition time and timeNext
          if(ignTime<ignMap[ncell]) ignMap[ncell] = ignTime;
          if( ignTime < timeNext )timeNext = ignTime;
          //...
```

Fig. 6. MT sequential pseudo-code fragment.

```
__global__ void FireKernel_MT( /*arguments*/){

//Associate thread to map element
cell = return_cell_idx

//Load ignition time to register
//to decrease global memory traffic (prefetching)
ignCell = ignMap[cell];

timeNext = //... do an atomic variable update
//Find burning cells
else if(ignCell == timeNow ){

  //Propagate from burning cells
  for (n = 0 ; n < 16; n++ ){
    ncell = //..compute neighbour cell index

    //Neighbour ign. time is pre-fetched
    ignNcell = ignMap[ncell];

    //If neighbour is unburned
    if( ignNcell > timeNow ){

      ROS = //...Compute ROS in the local - neighbour
            //direction (equation 1)
      ignTime = timeNow + L_n / ROS;

      ignMap[ncell] = //...atomic variable update
      timeNext      = //...atomic variable update
      //...
```

Fig. 7. MT kernel pseudo-code fragment.

$t^{n+1} = t^n + \Delta t$. The propagation part is once gain offloaded to the GPU.

The state parameter S of the algorithm (see Section 2.3) must be limited to the range [0, 1] during the calculation. This is achieved through an operation that guarantees the correct numerical value:

$$S^{n+1} = \left(S^{n+1} \leq 1\right) \times S^{n+1} + \left(S^{n+1} > 1\right) \times 1$$

For execution in the GPU, threads are bundled together into warps, each warp executing in a single processor of the device. If different threads in the same warp find different execution paths, *i.e.*, execution of an *if* type condition heralds a different result among some of the warp threads, execution becomes serialized. The mentioned procedure is thus necessary to circumvent this serialization, transforming the problematic “if” clause into an arithmetic operation and thus raising the parallel efficiency.

The ignition time of the cell is updated when $S^{n+1} = 1$, to a new time $t_{ig} = t^n + \Delta t$. Finally the state matrix is updated to $S^n = S^{n+1}$ and another time iteration is performed until the stop criteria is satisfied.

4.3. Iterative Minimal Time, sequential implementation

The Iterative Minimal Time (IMT) is a novel algorithm, aimed at exploiting massive parallel architectures like those of the GPU, and obtaining the same results from the previously mentioned Minimal Time fire growth model.

As stated in Section 2, the main difference between IMT and the two previous fire growth models is the replacement of the time progression loop by a numerical iterative procedure, associated to the computation of a residue. A kernel is launched at each iteration to compute a new ignition map, until the difference between consecutive iterations is less than a pre-defined threshold. The IMT kernel pseudo-code is listed in Fig. 8.

A thread is assigned to each cell to perform the neighbour operations (Equation (3)) and obtain the minimum ignition time need for each iteration.

To avoid divergence within the warp threads and possible serialization, the conditional statement present in the sequential version is replaced by function akin to that described in the previous model, trading divergence for more arithmetic operations.

This novel algorithm yields superior speedups because there is no divergence at the warp level and because a higher ratio between operations and memory access is achieved. Also, similarly to the finite time algorithm, fire propagates to the centre of each cell, guaranteeing absence of data races, an ideal property for massive parallel processing.

5. Results

Each model was implemented in a sequential and a parallel GPU version. Tests were run in an AMD Opteron(tm) processor at 2.4 Ghz and an Intel Xeon at 2.5 Ghz with a Nvidia Tesla C1060. Operating System was Linux (Debian), with kernel version 2.6.20. All CPU versions were compiled using gcc (Gnu Compiler Collection).

5.1. Verification

As already stated, this paper focus is on fire front prediction algorithms that form the base of current, in use, tools of computer-aided decision for fire fighting scenarios. The paper goal is to provide meaningful speedups to such tools using a novel, albeit almost ubiquitous, hardware platform for general computing, *i.e.*, the GPU. In this sense the models implemented have already been target of extensive validation in the literature and the authors refer to, for a validation of pure cellular automata

```
__global__ void FireKernel_IMT(/*arguments*/){

  //Associate thread to map element
  cell = return_cell_idx

  //pre-fetch cell's ignition time
  ignCell = ignMap[cell];
  //Skip 1st igniting cells
  if (ignCell > 0){
    ignTime_min = INF;
    //Loop cell neighbours
    for (n = 0; n < 16; n++){
      ncell = //...compute neighbour cell index

      ROS =  //...Compute ROS in neighbour-local direction
             //with equation 1

             //compute cell ignition time
     ignTime = ignMap[ncell] + L_n/ROS;

      //Find minimum ignition time with algebraic procedure
      ignTime_min = ignTime*(ignTime < ignTime_min)
                  + ignTime_min*(ignTime_sh >= ignTime_min);
    }
    //update k+1 ignition map
    ignMap_new[cell] = ignTime_min;
  }
}
```

Fig. 8. IMT kernel pseudo-code fragment.

models (Trunfio, 2004; Berjak and Hearne, 2002). For Raster fire modelling to FireStation (Lopes et al., 2002) and BFOLDS (Perera et al., 2008).

Synthetic test cases, with know exact solutions, where run, enabling error analysis, comparison and verification against those reported in the literature (see, for instance, *e.g.* Cui and Perera, 2008).

Two of the verification cases used are shown for purpose of illustration. Case A represents a no wind, flat terrain situation (Fig. 9a) and case B constant wind and slope (Fig. 9b). As expected, both minimal time models exhibit better results than the fixed time step model. For the elliptical situation shown in Fig. 9b the error increases in the direction of maximum propagation. It was found that the shape predicted by the models, for either the presented cases and other configurations tested but not shown, was in accordance to those found reported in literature (see, for instance, *e.g.* Cui and Perera, 2008).

Simulation errors in the fire growth models arise from three major sources:

i availability and quality of input data;
ii theoretical basis of the fire behaviour model;
iii fire growth algorithm.

The comparison between the exact solution, Minimal Time and Fixed Time, models eliminates the first two error sources. Comparisons of predicted shapes and sizes for simulated fire perimeters against their exact solution under 10 spatial resolutions, two wind directions and keeping constant the fuel type, weather, slope and aspect, where made. It was found that using finer spatial resolutions does not necessarily translate into more accurate predictions. This result agrees with those of Trunfio (2004), Finney (1998) and Cui and Perera (2008), that have shown that, for instance, the number of stencil neighbours, the number of directions for fire spreading and the raster shape play a larger role in achieving more accurate predictions.

Simulation errors occur mainly in the direction of the fire head, as seen in Fig. 9b. In this direction the ROS is larger, and so are the errors in the spread distance. The shape distortion errors seen in the Minimal Time model (Fig. 9a and b) arise from the fixed number of regular pathways for fire travel, as can be seen by the stencil in Fig. 1. The pathways are defined by the line connecting the ignited cell centroid to those of its neighbours.

In the Minimal Time model the fire will propagate from an ignited cell to its 16 neighbour cells according to Fig. 1. Ignition time for an unburned neighbour cell is given by the distance connecting both cells divided by the ROS in that direction. Evaluating the ignition time of all neighbours, the fire will propagate towards the neighbour with the lowest time to ignite. On the Fixed Time algorithm fire occurrence results from an heuristic average relating the eight stencil neighbours. The different rules used in the Minimal Time and Fixed Time models result in different predictions for the fire perimeter, as seen in Fig. 9a and b. Likewise to the Minimal Time model, larger errors in the main fire propagation direction are also found in the Fixed Time model.

5.2. Performance

The performance of the GPU implementation of each model was assessed using complex terrain conditions, to increase the realism of the results. Fig. 10 shows the height map and fire front predictions.

One of the main performance indicators of the GPU is its occupancy, defined as the ratio of threads running in the GPU streaming multiprocessor and the maximum number of threads it can accommodate. A higher occupancy value is generally desirable for good performance. One of the parameters influencing the occupancy is the block size, *i.e.*, in CUDA threads are bundled into warps which are bundled into blocks (see Kirk and Hwu, 2009).

A study of performance variation with block size was done for the different implementations. Fig. 11a, b and c shows the speedup for several block dimensions in different models. While the MT displays better performance with smaller blocks, both the Fixed Time and IMT behave differently, having better results for 256 threads per block (16×16).

For each case, the occupancy can be obtained using the calculator made available at the Nvidia CUDA website. Input parameters are the register use per thread, the shared memory per block, and the block dimensions. The first two are know at compile time, as part of the compilation log and the last, the block size, is user

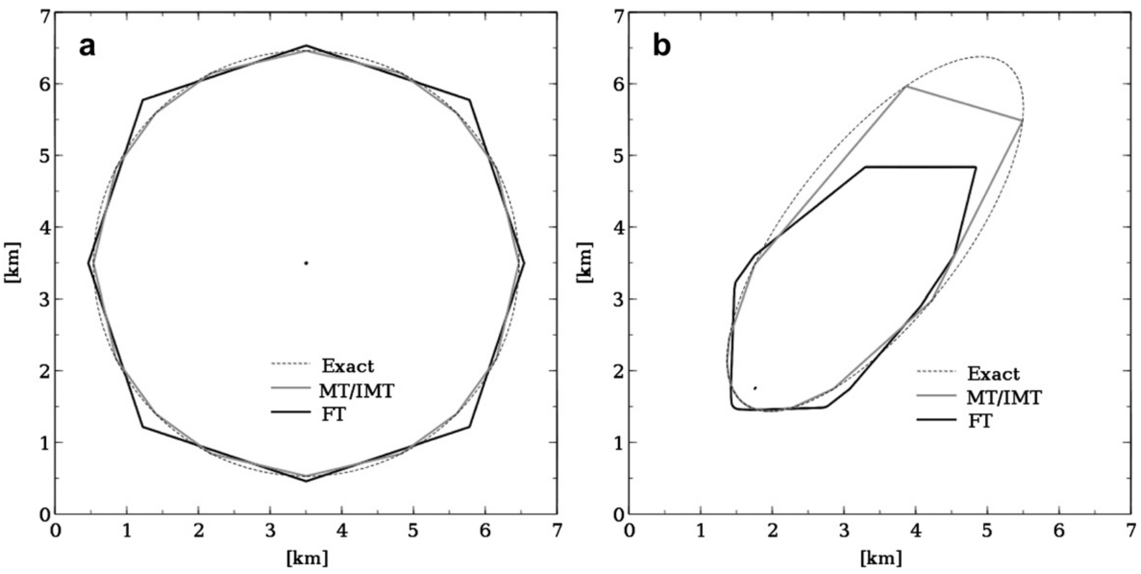

Fig. 9. Verification results for two constant condition scenarios. 1024^2 cells. (a) 5% Moisture, fuel model NFFL 1 (Andrews, 1986), no wind, no slope. Fire line at 2000 min. (b) 5% Moisture, fuel model: NFFL 1 (Andrews, 1986), Wind speed/direction: 8 km h^{-1} NE, slope/aspect: 0.5/23° clockwise from south. Fire line at 120 min.

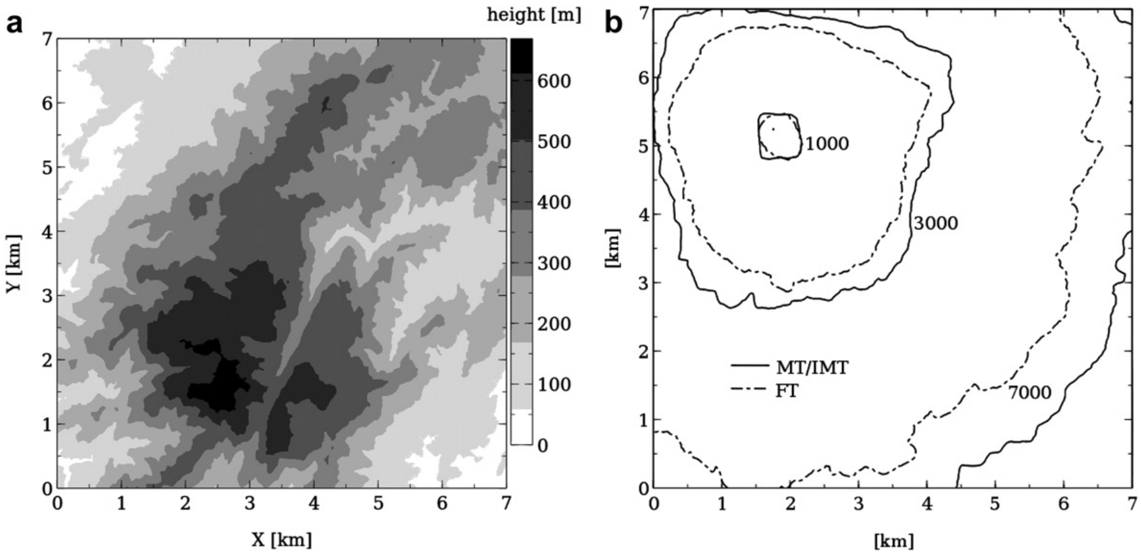

Fig. 10. Map and example results of a more realistic case for performance assessment. (a) Height map in metres for realistic test case. (b) 10% Moisture, fuel model NFFL 1, no wind, 1024^2 cells ($\Delta t = 1$ s for FT model). Contour plots in minutes.

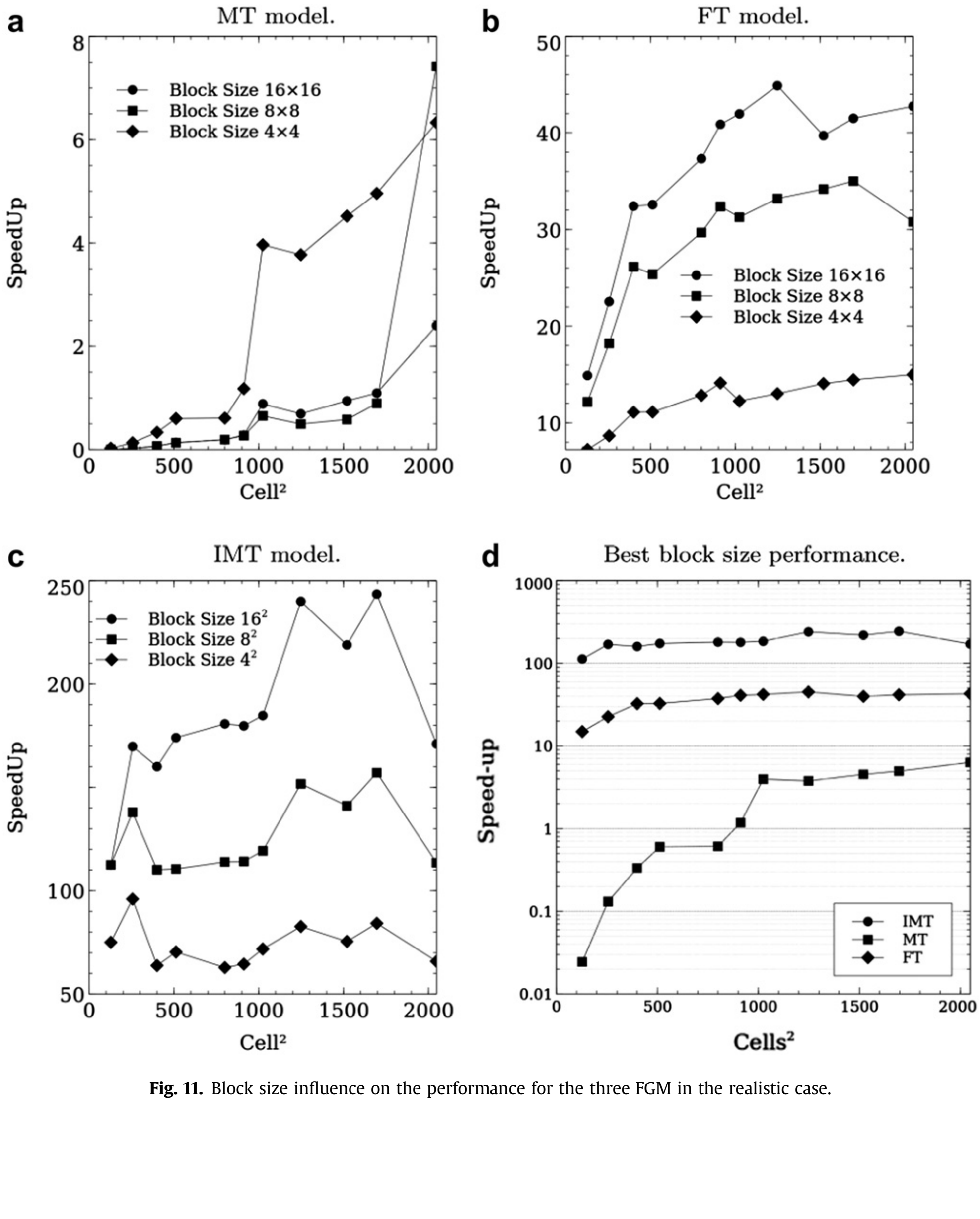

Fig. 11. Block size influence on the performance for the three FGM in the realistic case.

Table 1
Occupancy ratios for the main kernels of the CUDA implementations, in function of several block sizes.

		MT16 model	FT Model	IMT16
Registers		21	20	16
sh. mem (bytes)		872	76	3156
Occ. %	BS 4^2	25	25	25
	BS 8^2	50	50	50
	BS 16^2	50	75	100

defined. Because the majority of the computing time is spent on the fire propagation kernel, the efficiency and speedups of the CUDA programs should be coherent with the occupancy rates of the kernels. This data is listed in Table 1.

Fig. 11b and c follow exactly this mechanism, with each block size curve corresponding to each occupancy value, *i.e.*, BS 16^2 has occupancy = 75% and 100%, thus, the higher performances, BS 4^2 has occupancy = 25% and the worst performance and BS 8^2, with occupancy = 50%, fits exactly in the middle.

However, the MT model shows different results, with higher performance for smaller blocks. In this case, while larger blocks result in higher occupancy, smaller ones help reduce serialization when one thread stalls. The memory locks used in the MT model to deal with race conditions will make several threads stall and, as a consequence, the thread block will also stall in the SM. Smaller blocks will have less threads stalling and so present a higher performance for this situation.

Fig. 11d shows the speed-up curves for the cases with the most efficient block sizes. Relatively to other models, MT is the most memory demanding algorithm and has the worst speed-up results. The motive is, once again, the serialization introduced by memory locks. Maximum speed-up is achieved at 2048^2 cells with 6.3 $\times$ and the minimum value is 0.02 $\times$ at 128^2 cells, *i.e.* the GPU implementation is actually slower than the CPU version. Actually, 1024^2 cells are required for the GPU to outperform the CPU implementation of the MT model.

The Fixed Time model is not as memory demanding as MT, thus presenting superior gains. However, it still suffers from warp divergence. Maximum and minimum speed-up values are 44.9 and 14.9$\times$, for 1248^2 and 128^2 cells respectively. The speed-up results for the novel IMT algorithm proves that all aspects that were hindering performance on the GPU were dealt with. Eliminating warp level divergence and presenting a higher ratio between processor operations and memory bandwidth achieved a speed-up range between a maximum value of 243.5$\times$ for 1696^2 cells and a minimum value of 112.5 for 128^2 cells.

This performance opens the door to fine grain resolution and/or stochastic calculations, prohibitive under the current and predictable generations of serial CPUs.

Stochastic methods allow to gauge the influence of the uncertainty present in the initial data input of fire simulations, *e.g.*, velocity fields, fire positions, fuel, etc. This added information will benefit fire management teams during the fire fight, enhancing existing models with probabilistic results. A publication will be put forward about faster-than-real stochastic fire prediction (see Sousa et al., under review).

6. Conclusion

Two existing and known fire growth models of the raster type were successfully ported to the GPU architecture using CUDA.

The Fixed Time algorithm was used to raise the ratio of operations per memory accesses. Although an increase in speed-up was achieved, this algorithm has shown a degradation in the fire prediction capabilities of the code when compared with the Minimal Time fire growth model contagious algorithm. Coupling both algorithms, a novel approach was developed, joining the computational load of the Fixed Time algorithm with the fire prediction capability of the Minimal Time algorithm. The resulting algorithm, Iterative Minimal Time, is simple and straightforward, delivering two orders of magnitude speedups.

The developed work highlights that the best results were attained with a purposed tailored algorithm approach, instead of fine-tuning the kernel parameters. The speedups attained allow faster-than-real time predictions of fire behaviour and will allow stochastic modelling approaches, either Monte Carlo or other techniques, that require hundreds or thousands of runs, during a characteristic time interval of fire spread, *e.g.* 1 h of real time fire evolution.

All the source code is available under request to the authors, being licensed under GPL V3.

Acknowledgements

The authors would like to acknowledge Dr. José M. C. Pereira for the fruitful discussions on fire modelling and particularly on firelib.

References

Anderson, D.H., Catchpole, E.A., De Mestre, N.J., Parkes, T., 1982. Modelling the spread of grass fires. The ANZIAM Journal 23 (04), 451–466.
Andrews, P.L., Chase, C.H., 1989. BEHAVE: Fire Behavior Prediction and Fuel Modeling System – Burn Subsystem, Part2. USDA Forest Service, Intermountain Forest and Range Research Station, General Technical Report INT-260, 93 pp.
Andrews, P.L., 1986. BEHAVE: Fire Behavior Prediction and Fuel Modeling System – Burn Subsystem, Part1. USDA Forest Service, Intermountain Forest and Range Research Station, General Technical Report INT-194, 130 pp.
Bell, N., Garland, M., Dec. 2008. Efficient Sparse Matrix-vector Multiplication on CUDA. NVIDIA Technical Report NVR-2008-004. NVIDIA Corporation.
Berjak, S.G., Hearne, J.W., 2002. An improved cellular automaton model for simulating fire in a spatially heterogeneous savanna system. Ecological Modelling 148 (2), 133–151.
Bolz, J., Farmer, I., Grinspun, E., Schröder, P., 2003. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. ACM Transactions on Graphics 22, 917–924.
Burgan, R.E., Rothermel, R.C., 1984. BEHAVE: Fire Behavior Prediction and Fuel Modeling System – FUEL Subsystem. USDA Forest Service, Intermountain Forest and Range Research Station, 126 pp.
Cohen, J.M., Molemaker, J.M., 2009. A fast double precision CFD code using CUDA. Proceedings of Parallel Computational Fluid Dynamics.
Costa, A.M., Pereira, J.C.F., Siqueira, M., 1995. Numerical prediction of fire spread over vegetation in arbitrary 3d terrain. Fire and Materials 19 (6), 265–273.
Cui, W., Perera, A.H., 2008. A Study of Simulation Errors Caused by Algorithms of Forest Fire Growth Models. Ontario Forest Research Institute, Sault Ste. Marie, ON, Forest Research Report No. 167. 17 p.
Elsen, E., LeGresley, P., Darve, E., 2008. Large calculation of the flow over a hypersonic vehicle using a gpu. Journal of Computational Physics 227 (24), 10148–10161.
Encinas, A.H., Encinas, L.H., White, S.H., Rey, A.M., Sánchez, G.R., 2007. Simulation of forest fire fronts using cellular automata. Advances in Engineering Software 38 (6), 372–378.
Finney, M.A., 1998. FARSITE: Fire Area Simulator – Model Development and Evaluation. USDA Forest Service. Rocky Mountain Research Station Research Paper RMRS-RP-4 Revised.
Finney, M.A., 2006. An overview of flammap fire modeling capabilities. In: Andrews, P.L., Butler, B.W. (Eds.), Fuels Management – How to Measure Success: Conference Proceedings. U.S. Department of Agriculture, Forest Service, Rocky Mountain Research Station, Fort Collins, CO, pp. 213–220. Portland, OR. Proceedings RMRS-P-41.
Habich, J., July 2008. Performance Evaluation of Numeric Compute Kernels on NVIDIA GPUs.
Innocenti, E., Bernardi, F., Muzy, A., Capocchi, L., Santucci, J.-F., 2005. Distributed Fire Spreading Simulation Using Openmosix.
Innocenti, E., Silvani, X., Muzy, A., Hill, D.R., 2009. A software framework for fine grain parallelization of cellular models with openmp: application to fire spread. Environmental Modelling & Software 24 (7), 819–831.
Jorba, J., Margalef, T., Luque, E., 2001. Simulation of forest fire propagation on parallel & distributed PVM platforms. In: Cotronis, Y., Dongarra, J. (Eds.), Recent Advances in Parallel Virtual Machine and Message Passing Interface. Lecture Notes in Computer Science, vol. 2131. Springer, Berlin/Heidelberg, pp. 386–392.
Karafyllidis, I., Thanailakis, A., 1997. A model for predicting forest fire spreading using cellular automata. Ecological Modelling 99 (1), 87–97.

Kirk, D.B., Hwu, W.-M.W., 2009. Programming Massively Parallel Processors: a Hands-on Approach, first ed. Elsevier Morgan Kaufmann.

Larini, M., Giroud, F., Porterie, B., Loraud, J.-C., 1998. A multiphase formulation for fire propagation in heterogeneous combustible media. International Journal of Heat and Mass Transfer 41 (6–7), 881–897.

Lopes, A.M.G., Cruz, M.G., Viegas, D.X., 2002. Firestation – an integrated software system for the numerical simulation of fire spread on complex topography. Environmental Modelling and Software 17 (3), 269–285.

McDonough, J.M., Garzón, V.E., 1999. Parallel simulation of wildland fire spread. In: Lin, et al. (Eds.), Parallel Computational Fluid Dynamics. North-Holland Elsevier, pp. 101–108.

Mell, W., Jenkins, M.A., Gould, J., Cheney, P., 2007. A physics-based approach to modelling grassland fires. International Journal of Wildland Fire 16, 1–22.

Moore, G.E., April 1965. Cramming more components onto integrated circuits. Electronics 38 (8), 114–117.

Morvan, D., Dupuy, J.L., 2004. Modeling the propagation of a wildfire through a mediterranean shrub using a multiphase formulation. Combustion and Flame 138 (3), 199–210.

NVIDIA, 2009. NVIDIA CUDA Programming Guide, second ed.

Papadopoulos, G.D., Pavlidou, F.-N., 2011. A comparative review on wildfire simulators. IEEE Systems Journal 5 (2), 233–243.

Pastor, E., Zárate, L., Planas, E., Arnaldos, J., 2003. Mathematical models and calculation systems for the study of wildland fire behaviour. Progress in Energy and Combustion Science 29 (2), 139–153.

Perera, A.H., Ouellette, M., Cui, W., Drescher, M., Boychuk, D., 2008. BFOLDS 1.0: a Spatial Simulation Model for Exploring Large Scale Fire Regimes and Succession in Boreal Forest Landscapes. Ontario Forest Research Institute, 50 pp.

Phillips, E.H., Zhang, Y., Davis, R.L., Owens, J.D., Jan. 2009. Rapid aerodynamic performance prediction on a cluster of graphics processing units. In: Proceedings of the 47th AIAA Aerospace Sciences Meeting. No. AIAA 2009-2565.

Rodriguez, R., Cortes, A., Margalef, T., 2010. Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on Data Injection at Execution Time in Grid Environments Using Dynamic Data Driven Application System for Wildland Fire Spread Prediction. pp. 565–568.

Rothermel, R., 1972. A Mathematical Model for Predicting Fire Spread in Wildland Fuels. U.S. Department of Agriculture, Forest Service, Intermountain Forest and Range Experiment Station. Res. Pap. INT-115, Ogden, UT.

Scott, J.H., Burgan, R.E., 2005. Standard Fire Behavior Fuel Models: a Comprehensive Set for Use with Rothermel's Surface Fire Spread Model. Gen. Tech. Rep. RMRS-GTR-153. U.S. Department of Agriculture, Forest Service, Rocky Mountain Research Station, Fort Collins, CO. 72 p.

Sousa, F., Ervilha, A.R., Pereira, J.M.C., Pereira, J.C.F., under review. Faster Than Real Time Stochastic Fire Spread Simulations.

Sullivan, A.L., 2009a. Wildland surface fire spread modelling, 1990–2007. 2: empirical and quasi-empirical models. International Journal of Wildland Fire 18, 369–386.

Sullivan, A.L., 2009b. Wildland surface fire spread modelling, 1990–2007. 3: simulation and mathematical analogue models. International Journal of Wildland Fire 18, 387–403.

Thibault, J.C., Senocak, I., Jan. 2009. Cuda implementation of a Navier–Stokes solver on multi-gpu desktop platforms for incompressible flows. In: Proceedings of the 47th AIAA Aerospace Sciences Meeting. No. AIAA 2009-758.

Tolke, J., Krafczyk, M., 2008. Teraflop computing on a desktop pc with gpus for 3d cfd. International Journal of Computational Fluid Dynamics 22 (7), 443–456.

Trunfio, G.A., 2004. Predicting wildfire spreading through a hexagonal cellular automata model. In: Sloot, P.M.A., Chopard, B., Hoekstra, A.G. (Eds.), Cellular Automata. Lecture Notes in Computer Science, vol. 3305. Springer, Berlin/Heidelberg, pp. 385–394.

Viegas, D.X., 1998. Forest fire propagation. Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences 356 (1748), 2907–2928.

Wiggers, W., Bakker, V., Kokkeler, A., Smit, G., November 2007. Implementing the conjugate gradient algorithm on multi-core systems. In: Nurmi, J., Takala, J., Vainio, O. (Eds.), Proceedings of the International Symposium on System-on-Chip (SoC 2007). No. 07ex1846. IEEE, Piscataway, NJ, pp. 11–14.

Zhou, X.Y., Pereira, J.C.F., 2000. A multidimensional model for simulating vegetation fire spread using a porous media sub-model. Fire and Materials 24 (1), 37–43.