Highly Parallel Implementation of Forest Fire Propagation Models on the GPU

Jessica Smith*, Lee Barford*†, Sergiu M. Dascalu* and Frederick C. Harris, Jr.*
*Department of Computer Science and Engineering, University of Nevada, Reno, USA
†Keysight Laboratories, Keysight Technologies, USA
jsmith@nevada.unr.edu, lee_barford@ieee.org, dascalus@cse.unr.edu, Fred.Harris@cse.unr.edu

***Abstract*—Forest fire simulation is a challenging, complex problem which requires large amounts of data to be processed for an accurate simulation. This paper implements Rothermel's fire spread equations on three different spread methodologies. Both a sequential implementation of each spread methodology along with a parallel implementation are presented in this paper. The parallel version is implemented on the Graphics Processing Unit (GPU) using NVIDIA's CUDA programming language. The parallel spread methods achieved runtimes in those ranges realistic for fighting a real-time forest fire. The GPU implementation achieved faster running times in each of the spread methodologies, ranging from 64x to 229x faster than the sequential implementation.**

I. Introduction

Every year, fighting forest fires costs taxpayers in the United States millions of dollars. From 2002 to 2012, the average amount spent per year on forest fire suppression by the federal government was $962 million, which only amounted to 32% of the entire federal wildfire protection funds [1]. This cost only covers the federal funds that are spent on fighting and preventing forest fires. It does not include the individual and environmental cost of loss of property and habitat. The highest cost that occurs during the efforts to fight a forest fire is the loss of life incurred by fire fighters. The ability to better predict the behavior of a wildfire greatly increases the effectiveness of firefighting efforts, thereby reducing all the costs incurred during a forest fire. Another application domain for forest fires is the training of fire fighters. A sample forest fire could be provided in training scenarios which would allow fire fighters to have as close to hands-on forest fire training before they are exposed to real wildfires.

In order to simulate wildfires, scientists have developed several methods for modeling the propagation of fire [2][3]. These fire models are based on the properties of the environment in which the forest fire takes place. Such properties include but are not limited to, fuel load, fuel type, wind, live moisture, dead moisture, and crown height. Incorporating these and other variables into spread models can allow for accurate prediction of where a fire will spread and how quickly it will arrive. Research into developing fuel and moisture models is active to this day. These models provide the basis for the properties on which forest fire simulation is based.

The ability to realistically simulate forest fires is desirable because it allows fire experts to more accurately predict the impact of their fire-fighting decisions. Possible manipulations to the wildfire environment include adjusting moisture content to simulate a water drop, adjusting fuel loads where a simulated bulldozed treeline could exist, or reverse spread testing in which a fire started by firefighters would burn the fuel away from the advancing wildfire. Unfortunately, the amount of data required for realistic fire simulations requires a large amount of computation time to produce an accurate simulation. The more accurate and fine-grained the simulation, the longer it takes to process the data. A forest fire is a dynamic entity, therefore the ability of a simulator to run in real time is necessary for it to be an effective tool. The more complex and accurate a simulator is, the more useful it is to fire scientists. There are multiple different aspects to accurately modeling the spread of a forest fire. The main four fire properties which influence the spread of a fire are base fires, crown fires, fire acceleration, and spotting [4]. Base fire spread is addressed in this paper, and is the basis for all fire spread simulations along the floor of a forest. Crown fires occur when the forest fire spreads into the tops of the trees in a forest. A crown fire may be passive or active. An active crown fire is one which contributes to the overall spread of the fire. A passive crown fire will burn in the tops of the trees but is not hot enough to contribute to the overall spread of the fire. Fire acceleration is the phenomena that accounts for the dynamic speed of a fire. A forest fire will not automatically spread at the maximum rate for which it has the potential. Spotting is the phenomena of fire embers being blown forward ahead of the fire to ignite new fires in zones isolated from the main body of the fire. The implementation of these four main aspects to forest fires builds a state of the art forest fire simulator. This paper only focuses on implementing the base fire spread case, and porting it to the GPU.

Using the GPU as a general purpose computing device has become popular in recent years, especially on problems which require a large amount of data processing. The GPU is ideally suited to high volume data processing applications because it can processes millions of inputs simultaneously, while a CPU may process only one or a few at a time [5]. This paper implements three fire spread methods based on Rothermel's fire spread equations on both the CPU and GPU, and compares the resulting run times, throughput, and accuracy.

The remainder of the paper is structured as follows. Section II describes related work. Section III describes the fire propagation models implemented in this paper. Section IV covers the sequential algorithms used as a baseline for comparison in the paper. Section V describes the parallel implementation on the GPU of the fire spread algorithms. The experimental results of this paper follow in Section VI.

II. Related Work

In order to build a forest fire simulator, forest specific data must be integrated. Rothermel developed the first eleven fuel models that are still used to this day [2]. The method by which these eleven fuel models were created is the basis for the development of all modern fuel models. The fuel model contains information on properties of the forest in a particular region, and at a certain granularity. The forest is broken up into cells, each cell having properties which are modeled in the fuel model. For example, one fuel model might describe a coniferous forest in regions of 30x30 meter cells. These cells are what make up the basis for a fire simulation.

The majority of the existing forest fire simulators, including this work, calculate wildfire spread based on the Rothermel's fire spread equations [2]. More detail on the simulators which use this fire spread model will be covered later in the section. Equation 1 shows his rate of spread equation, which is based on several parameters.

$$R = \frac{(I_p)_o(1 + \phi_w + \phi_s)}{\rho_b \varepsilon Q_{ig}} \quad (1)$$

Where R is the rate of spread, $(I_p)_o$ is the no-wind propagating flux, ϕ_w and ϕ_s are the additional propagating flux introduced by wind and slope respectively. The product of ρ_b and ε is referred to as the effective bulk density. The effective bulk density models the amount of fuel per unit volume of the fuel bed raised to ignition ahead of the advancing fire. Q_{ig} is the heat of preignition (the heat required to bring a unit weight of fuel to ignition). These values are derived or contained in the fuel model that describes the cell for which the computations are being done.

The desired output from a forest fire simulator is a time of arrival map. Each cell in this time of arrival map represents a cell in the simulation forest, and the value it contains is the time at which the cell ignited and started propagating the fire. Once a cell is lit, it begins contributing to the spread of the fire to the surrounding cells and continues burning until the fuel in that cell is entirely used. The method of propagation may vary between simulators, but the basic spread rate is usually based on Rothermel's equations. This paper addresses three such spread methods.

Since Rothermel's paper was published in 1972, several fire spread simulators have been developed. Every major forest fire simulator has used his spread equations as the basis for their simulation. While there have been new models developed for the spread of fire [4][6], the existing forest fire simulators use Rothermel's because it is an approximation of the spread of fire accurate and simple enough to be computable in the time allowed for simulations. The first major fire spread simulator was developed in 1986 called BEHAVE[3]. BEHAVE had two main functions to the application. The first function allowed users to load in fuel models from Rothermel's paper, but also to develop and save new fuel models. The simulator then had the ability to integrate the newly developed fuel models in its simulations. The second function of the application would run a simulation and burn prediction on the desired fuel model. The output of this simulator appeared in a table which represented the times of arrival for each cell in the simulation. There was no visualization method available for this simulator. The simulation was meant to be used as a training tool rather than a real-time tool to be used to fight wildfires.

A decade later in 1996, BEHAVE was the basis for a new forest fire library that was developed using C called fireLib [7]. The code was based entirely on BEHAVE's simulator, but brought up to a then-modern platform. FireLib can run much faster than BEHAVE, and the output is given in time of arrival arrays rather than a table. Each (x,y) in the array corresponds to a cell in the fire simulation, and the time of arrival is the time at which the cell ignites, and can then begin to propagate the fire. The fire library is more flexible than BEHAVE and allows a user to design their own methods for propagating the fire. There are a few different methods which may be used, and will be addressed later in the paper. However, where BEHAVE was an entire application which had an interface component, fireLib is simply an open source forest fire library and both the visualization and interface development are left up to the user.

In 2004, FARSITE was developed, which works as a full-scale forest fire simulator [8]. It has been continuously developed since 2004 and is currently still operational. It incorporates more features than simple fire spread, such as crown fires, surface fires, fire acceleration, and spotting. While it is one of the most advanced and accurate forest fire simulators, it is not very fast. It is one of the most widely used forest fire simulators in existence today.

This paper used much of the fire spread implemenation from a forest fire simulator called vFire [9]. vFire was based on hFire, and are both cellular based spread models. They run faster than FARSITE, but do not have the same level of precision [10]. vFire uses a technique that has dynamic time stepping to burn distances between cells to determine an accurate time of arrival for the fire spread. The important feature that vFire accomplished was porting the computation of the fire spread to the GPU using OpenGL shaders [11]. Because the computation was ported to the GPU, it accomplished a very high speedup over its sequential implementation. vFire provided the outline for the data processing portion of this work as the code was available to this project.

Sousa, dos Reis, and Pereira also used the GPU to improve their running times and ported fireLib to the GPU [12], but were the first to use the parallel programming language CUDA [13]. They implemented three kernels in which they explored three different propagation types. This paper based two of the spread methods (Minimal Time and Iterative Minimal Time) on the work done by Sousa, dos Reis, and Pereira. Their work will be covered in more detail further in the Fire Propagation Models section of the paper. The third propagation method implemented in this paper was based on work found in vFire [9].

Investigations into using GPU computation for optimizing fire simulation have been explored, including a paper by Arcaa, Ghisub, and Trunfioc [14]. Their implementation used GPU computation to optimize fuel treatments across a landscape. Their focus was not on using the GPU to calculate base propagation, rather the influence of fire breaks. The work by Baranovskiy explores the usage of GPU computing to enhance the performance of theoretical-based propagation models [15].

While the work is useful for exploring the usage of theoretical models, this work implements a semi-empirical approach, and therefore the direct results are not comparable.

III. Fire Propagation Models

There are several potential approaches to calculating the propagation of fire in a wildfire environment. This paper implemented three methods for iterating through a simulation to calculate the time of arrival map for a simple one-source forest fire under constant terrain and wind conditions. The first two spread methods (Minimal Time and Iterative Minimal Time) are based on stepping through time independent of specific fuel conditions and are based on the paper by Sousa, dos Reis, and Pereira [12]. The third spread method implemented in this paper (Burn Distances) was based on code and methods found in vFire [9]. vFire implemented an accurate spread rate calculator based on Rothermel's fire spread equations and the fire spread and fuel model data to propagate based upon the physical burning of fuel.

The model for propagation rate used in this paper was the same for all three spread methods. The model is based on Rothermel's fire spread equations, and is found in Equation 2. This equation was derived by the creators of vFire [9], and the data processing done in the preprocessing phase of this project was based on their work. The preprocessing step translates the equation from Rothermel's work seen in Equation 1 to what is seen in the following Equation 2. The preprocessing data is out of the scope of this paper and is not covered in detail.

$$r(\Theta) = R_{max} \frac{1.0 - \varepsilon}{1.0 - \varepsilon cos(\phi - \Theta)} \tag{2}$$

R_{max} is the maximum rate at which a fire can spread. ε is the eccentricity of the fire, which is based on wind and slope data. ϕ is the orientation. Θ is the direction in which the fire is spreading. R_{max}, ε, and ϕ are all computed based on the terrain data before the propagation simulation takes place. This is done in the preprocessing stage because the rates at each cell do not change until the forest model changes. An interactive simulator could potentially allow for these variables to change (i.e. modeling a water dump from a helicopter or a bulldozer tearing down a line of trees) but that is outside the scope of this research paper. To implement these features, changes would be made to the fuel and moisture models used to determine the possible rate of change in a cell. The direction Θ is computed based on which neighbor is being examined at the time.

During the preprocessing phase, there are several data files which need to be processed and interpolated to be of the same size. The fuel data and slope data are stored in files containing interpolation data such as size of cell, width, and height of the data grid. The Geospatial Data Abstraction Library (GDAL) was used to interpolate the data from the terrain and fuel files into the desired size of simulation [16]. Wind data is incorporated into the spread rate calculations as a 2D vector for each cell in the grid. The wind data contains a direction and magnitude. The fuel models provide the detailed parameters by which the rate of spread is calculated. There are potential areas in this processing phase (such as calculating the Rothermel spread properties) that could be parallelized to improve overall running times of this simulator. However, the focus of this paper was to explore the potential for calculating fire spread on the GPU, so these possibilities were not addressed and are left for future work.

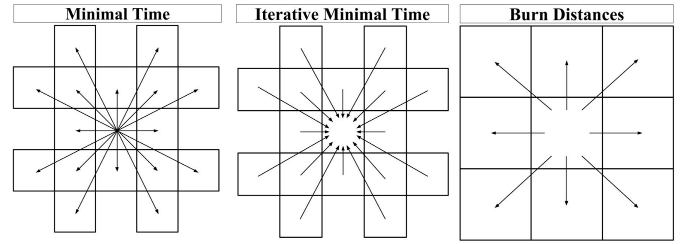


Fig. 1. Neighbor access methodology for each of the propagation methods. From left to right: Minimal Time, Iterative Minimal Time, Burn Distances

IV. Sequential Implementation

In order to accurately compare the performance of the GPU implementation of the fire spread simulations, a sequential version was implemented. The sequential implementation in this paper did not use the multi-threading capabilities of the CPU, and ran it in a purely sequential manner. The preprocessing steps are exactly the same for the sequential and parallel implementations, and are therefore not included in the timing and throughput comparisons. For this paper, the terrain was flat and the wind was set to zero to produce a general simulation of how basic fire would spread. The preprocessing calculations took less than 10% of the run time of the entire algorithm in small cases, and less than 2% in the large cases. The simulation stepping was the largest computationally and run time expensive portion of the simulator, and is the focus of this paper.

Once the preprocessing is complete, one of three simulation methods was applied: Minimal Time, Iterative Minimal Time or Burn Distances. The pseudo code for each kernel is found in Section IV-A through Section IV-C. The results for the sequential timings may be found in Figure 5(a). The worst of the timings was found to be in the Iterative Minimal Time method, taking nearly half an hour to run the large 2048x2048 simulation. Nearly all the sequential timings take an extensive amount of time. Since the goal of a forest fire simulator is to operate in real-time, it is unrealistic to expect to wait that long to receive the simulation data. After thirty minutes, the fire will have changed enough for the simulation's predictions to be irrelevant. Sequential running times for all algorithms may be seen in Figure 5(a). The following algorithm outlines the simulation steps from start to finish:

Algorithm 1 Simulation Progression

```
InitializeTerrainData();
CalculateSpreadRates();
RunPropagationSimulation();
GenerateOutputFile();
```

The InitializeTerrainData() and CalculateSpreadRates() functions in Algorithm 1 are part of the preprocessing in this project. These are not addressed in detail in this paper due to space constraints. The RunPropagationSimulation() portion is the focus of this paper and will be outlined in more detail in the following sections.

A. Minimal Time

The Minimal Time (MT) propagation method uses a dynamic time stepping method to step through the simulation. At each time step, a cell is examined to see if it is on fire. If the cell is burning, then the neighbors of the cell are examined as seen in Figure 1. If the neighbor is already on fire, it is ignored. A neighbor cell which has been lit during the current timestep, it is still examined for the event of a sooner time of arrival. If the neighbor is unlit, then the time of arrival for that cell is computed using the propagation equation found in Equation 2. In the Minimal Time method, time is incremented dynamically. Each time a new ignition happens, the ignition time is compared to the current 'timeNext' variable. If the new ignition time is smaller (the fire arrives sooner) than the current timeNext, then it replaces the timeNext value. This methodology means that time is incremented based on which cell will ignite the earliest. The pseudocode for the Minimal Time propagation method may be derived from Algorithm 2.

Algorithm 2 Minimal Time Algorithm

```
for cell = 0 to numCells do
  if timeNext > ignTime[cell] AND
    ignTime[cell] > timeNow then
      timeNext = ignTime[cell]
  else if ignTime[cell] == timeNow then
    // Propagate Fire
    for n = 0 to 15 do
      //If neighbor is unburned
      if ignTime[neighborCell] > timeNow then
        ROS = Compute ROS according to Equation 2
        ignTimeNew = timeNow + L_n/ROS
        if ignTimeNew < ignTime[neighborCell]
        then
          igntime[neighborCell] = ignTimeNew
        end if
        if ignTimeNew < timeNext then
          timeNext = ignTimeNew
        end if
      end if
    end for
  end if
end for
```

B. Iterative Minimal Time

The Iterative Minimal Time (IMT) method is designed to avoid data dependencies. Each cell operates independent of its neighbors, calculating the ignition time based on the spread rates, and only finishing when the values between step k and $k+1$ converge. Each cell looks at the minimal time for each of its neighboring cells to burn towards it as shown in Figure 1. The value is known to converge after the difference between two time steps is less than some small threshold. The most appropriate value for this threshold can be determined through experimentation. The pseudocode for the IMT spread method may be derived from Algorithm 3.

Algorithm 3 Iterative Minimal Time Algorithm

```
for cell = 0 to numCells do
  // Check for simulation completion:
  if |ignTime[cell] − ignTimeNext[cell]| < thresh then
    //Mark as converged
  end if
  if ignTime[cell] > 0 then
    ignTimeMin = INF
    //Propagate Fire
    for n = 0 to 15 do
      ROS = Compute ROS according to Equation 2
      ignTimeNew = timeNow + L_n/ROS
      ignTimeMin = MIN(ignTimeNew, ignTimeMin)
    end for
  end if
end for
```

C. Burn Distances

The Burn Distances (BD) method is based on the idea that it takes a certain amount of time for fire to burn the distance between two cells. This distance is set as equivalent between all cells and these properties are computed and loaded in the preprocessing stage of the program. Since they are handled in preprocessing, they are based on the properties of the forest model which is based on the size of the forest cell. The simulation iterates at a constant time step, and the amount the distance has burned is tracked throughout all the time steps.

An issue with this method arose from the static time step and needed to be addressed. Figure 2 shows the problem that can arise from the fixed time step. Figure 2 (a) shows the initial propagation. In this scenario, imagine that the propagation rate for b is much higher than a, but because the time step is large enough, they both propagate approximately one cell per time step. Figure 2 shows the second propagation step in time. In this example, a' propagates to a". Because of the faster propagation rate of b, b' should propagate to b" and b" to b"' before a' propagates to a". In the case where the time step is too large, the time of arrival would show an erroneous value for the cell holding a". To fix this issue, the time step in the simulation must be set to a small enough value to avoid this error. The time step should be set to the smallest possible time it takes fire to propagate from one cell to another.

A cell ignites when the distance from an ignited neighbor has been completely burned. The simulation checks a cells neighbors for ignition, and then uses their properties to burn the amount of distance in that time step towards the current cell as seen in Figure 1. The cell ignites when one of its neighbors burns the distance completely. The equation to determine how much distance is burned can be found in Equation 3.

$$d = d - r\Delta t \tag{3}$$

Where d is the distance that needs to be consumed. It is slowly decremented over time by the rate of spread (r) times the time step size (Δ t). In order to account for the fact that an 'overburn' could occur with fixed time steps. The exact time of arrival that is calculated is dependent on the exact time of arrival that the fire would have arrived at the cell. The equation used to find the exact time of arrival is found in Equation 4.

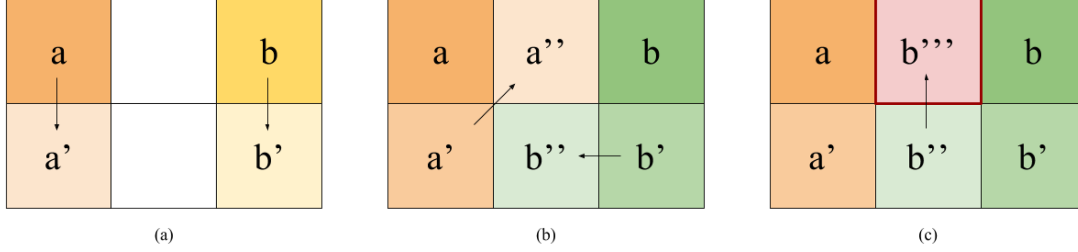


Fig. 2. The possible error for a fixed time-step propagation method. (a) shows the initial step with two lit cells a and b propagating to cells a’ and b’. (b) shows a’ propagating to a” in the next time step. (c) is the situation where b” would have propagated faster to the slot occupied in the previous step by a”, but because of the fixed time step, it would not propagate to an already lit cell.

$$TOA = t_{now} + \frac{d_{over}}{r} \tag{4}$$

Where TOA is the time of arrival that is written out to the time of arrival map, t_{now} is the time in the simulation during which the propagation is occurring, d_{over} is the distance the fire burnt past the desired difference, and r is again the rate of spread. The pseudocode for the algorithm for the BD propagation method is found in Algorithm 4.

Algorithm 4 Burn Distances Algorithm

```
for cell = 0 to numCells do
    // Check to see not on fire
    if ignTime[cell] == INF then
        Skip
    end if
    // Check Neighbors for fire propagation
    for n = 0 to 7 do
        if ignTime[neighborCell] < INF then
            Skip
        end if
        ROS = Compute ROS according to Equation 2
        burnDistance(totDist[neighborCell], ROS, timeStep)
        if distance is burnt then
            ignTime[neighborCell] = timeNow
        end if
    end for
end for
```

V. Parallel Implementation

As with the sequential implementation, the parallel implementation used the preprocessed terrain data to feed into the simulator. The kernels are written using the GPU programming language CUDA [13]. CUDA is a parallel computing programming language designed to run on NVIDIA produced GPUs. CUDA allows GPUs to be used for general-purpose computing, and is therefore the ideal language with which to program a highly parallelized version of the fire propagation model. Three kernels were implemented for this paper, based on the three propagation methods presented in Section IV. Each kernel was implemented to directly reflect the exact results of its sequential counterpart, meaning the two time of arrival maps matched exactly between sequential and parallel implementations. An overall algorithm for the kernel calls is outlined in Algorithm 5. The details on implementation of the kernels are found in the subsequent subsections.

Algorithm 5 Simulation Composition

```
while Simulation !Complete do
    computeKernel<<< Blocks, Threads >>>(inputs)
    terminateKernel<<< Blocks, Threads >>>(inputs)
end while
```

The blocks and threads referenced in Algorithm 5 are referencing the blocks and threads that are allocated by the GPU upon initiation of the kernel. Each GPU has a specific number of blocks that it can allocate, and each block has a number of threads it can allocate. The threads process the kernel code independently of each other. Two kernels are necessary in this work because there is no way to force block-wise synchronization without a kernel finishing its work. The CPU calls the two kernels in a loop in this manner to ensure that one time step’s propagation is calculated before the next can begin.

A. Minimal Time

The parallel implementation of the MT kernel is very similar to the sequential version, so due to this fact and space constraints, the pseudocode is not included in this paper. However, there are a few challenges which present themselves when implementing the algorithm. The first is data

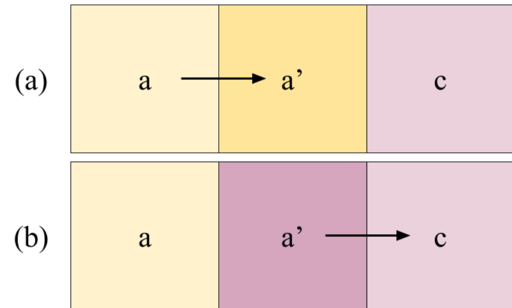


Fig. 3. This is an example of the problem syncing thread read access and write access. There is no way to stop one thread writing to another cell before the value is read by another cell.

synchronization. Each cell in the fire is processed by one thread at a time, but every thread need access to the time of arrival map for both reading and writing. A problem arises when one thread writes to a cell before another has the chance to read from it, which causes race conditions and simulation artifacts. In order to minimize these race conditions, atomic operations were used to ensure that data integrity is maintained. CUDA's AtomicMin() operation was used to ensure that a cell that was writing to a data location was not overwriting a smaller time of arrival, which is the correct value that needed to be stored [13]. Atomic operations in CUDA are designed to lock resources when a single thread is accessing them. The AtomicMin() operation ensures that the minimum of two integers is the one which is stored at the memory location [5]. This solves the problem of one thread reading a value for its comparison then another writing a value to that same location, which would mean the value against which the calculations are compared would be inaccurate. A visual example of this read-write problem can be seen in Figure 3. A secondary kernel was introduced to manage the time update between time steps in the simulation. In Algorithm 5, this secondary kernel accounts for the termination kernel. The secondary kernel was found to be the most efficient solution to this problem. CUDA does not allocate threads in any specific order, which means thread 1 could be the last to finish calculations while thread 1,000 could be the first [5]. In order to step through time in the MT propagation method, the timeNext variable needs to be set after all calculations finish. Since there is no way to ensure which thread would be the last to finish operating on the data, another method for iterating the variable was needed after all threads had finished their computation. The secondary kernel is called after the first terminates, which ensures that all threads have finished their computation before the timeNext variable is updated. Copying the two-element time vector back to the host device and managing the update there was also tested, but it was found to be faster to write a new kernel to update the data without copying anything back to the host device.

B. Iterative Minimal Time

The parallel implementation of the IMT kernel is also very similar to the sequential implementation, and for the same reasons as the MT kernel pseudocode is not included, neither is IMT's. The same data synchronization issues are also found in this kernel, where reading from and writing to the same cells would cause race conditions. A simple example of this problem may be seen in Figure 3. There is no way to stop a thread from writing to an output position before another thread reads in the data it needs to do its own spread calculations. Figure 3 (a) shows the writing of the value from a to a'. The value that existed before a' was the appropriate value for c to read in to do its calculations. Instead as seen in Figure 3 (b), it is the result value from a that it reads into do its calculations. This error causes race conditions to occur and artifacts to appear in the simulation. Instead of using atomic operations to solve this problem, two time of arrival vectors are passed to the kernel at startup. The two kernels act as an input and output kernel, reading from the former and writing to the latter. This introduces a new problem with maintaining accurate spread data across the input and output vectors. The secondary kernel in the IMT method is both used to copy data from the output back into the input as well as checking for the terminating condition for the simulation.

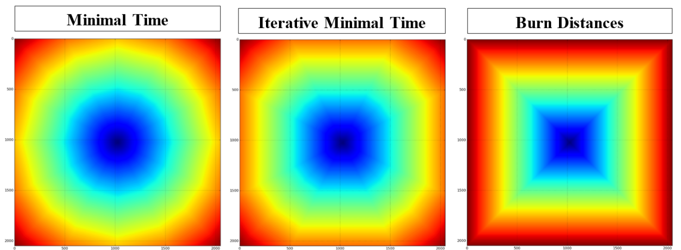


Fig. 4. The ignition maps produced by each propagation method. (From left to right: Minimal Time, Iterative Minimal Time, Burn Distances)

C. Burn Distances

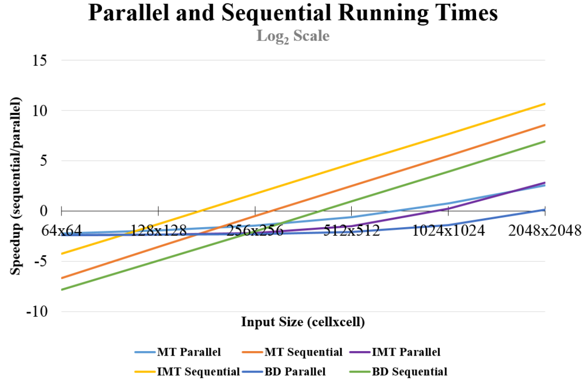
The parallel implementation of the BD kernel is also not included for the same reasons as MT and IMT's are not. The burn distances kernel faced the same data synchronization issue as IMT, and it is solved in the same manner with an input and output vector. The terminating kernel is also a copy kernel to eliminate the chance for read/write errors to and from the time of arrival map.

D. Properties of the Parallel Implementation

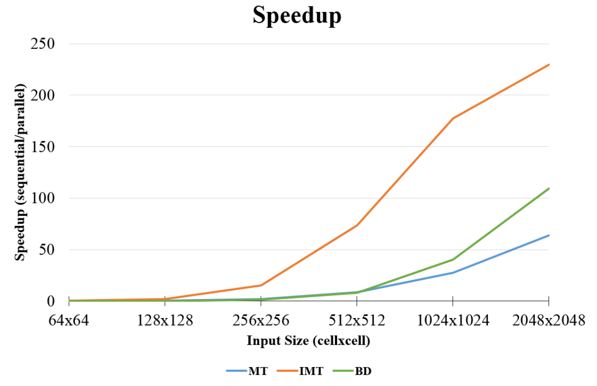
CUDA allows programmers to call kernels with a specific number of blocks and threads. The number of blocks and threads allocated will partially determine the running time of the program. For this project, the optimal number of blocks and threads were determined experimentally. For the MT method, it was found that a maximum block size of 1024 blocks was optimal, unless the input width was less than 1024, in which case a block size equal to the width of the simulation was optimal. In the MT implementation, 128 threads performed consistently better than any other configuration. For the IMT kernel, the optimal block size followed the same pattern as the MT method, but the optimal thread number was 256 instead. It was discovered that the optimal number of blocks for the BD method was the same as the width of the simulation, while the optimal thread number was 256. In all cases, the number of threads was set to the simulation width once it became smaller than the optimal thread amount. Until the size of inputs exceeds the maximum block and thread count, each cell in the simulation is processed by one and only one thread. In the case where one thread has to process multiple inputs, striding is used to accomplish the simulation.

VI. Results

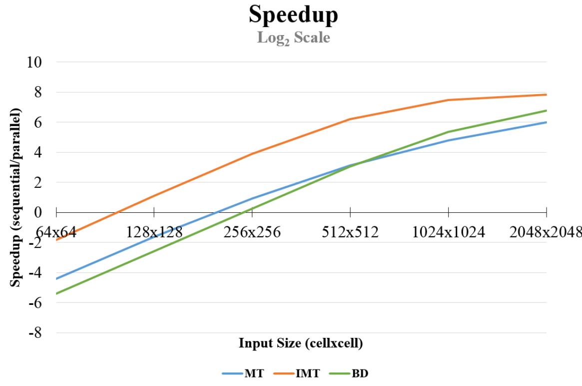
The results were determined by running the sequential and parallel versions of the code on the same machine. The result timings involved do not include the preprocessing times for the data, because those times are consistent across all implementations and do not significantly impact the total run time. The resulting time of arrival maps produced by the largest test input size (2048x2048) may be seen in Figure 4. The figure shows the different outputs received by the same preprocessed data for each of the propagation methods. Recall that the terrain is level and there are no winds impacting these simulations. Because the results in this paper were timed on different data and terrain values, with newer GPU iterations,



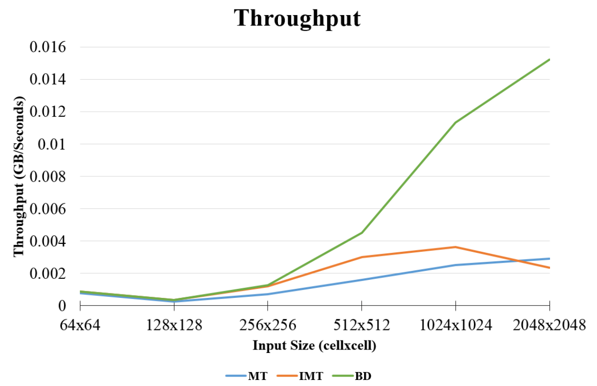
(a) Timings in seconds and log_2 scale for all the implementations.



(b) Speedup graph found by dividing GPU/CPU running times.



(c) Log_2 based graph of the speedup.



(d) The throughput found in GB/Seconds for the GPU kernels.

Fig. 5. All the result graphs for the MT, IMT, and BD kernels.

the results presented in Sousa, dos Reis, and Pereira's work are not comparible to these results directly [12]. Their code was not available to the public, and could not be tested under the same environmental constraints as this work and so their results are not compared. This section presents results comparing the sequential and parallel implementations accomplished by this work.

A. Hardware

Both the sequential and parallel implementations of this paper were run on a CUBIX box [17]. The sequential results were timed using an Intel(R) Xeon(R) CPU @ 2.00GHz, which has 6 cores and a cache size of 15MB. The parallel results were timed using a single NVIDIA GeForce GTX 780, which has 2304 cores and 3072 MB DDR3/GDDR5 of standard memory. Each of the tests were run five times, and the median value of the set is graphed.

B. Timings

In order to test the implementations of this project, several simulations were run. The simulations were sized from 64x64 cells to 2048x2048 in increasing powers of two. Figure 5(a) shows the sequential and GPU timings in a log_2 scale over all input sizes. A table of all running times may be found in Table I. Every kernel ran the largest simulation size in under 8 seconds, the BD performing the fastest at just over 1 second.

TABLE I. EXECUTION TIMES ON ALL RUNS

	Sequential (seconds)			Parallel (seconds)		
Cells	MT	IMT	BD	MT	IMT	BD
64	0.010	0.053	0.004	0.210	0.187	0.189
128	0.085	0.409	0.032	0.259	0.193	0.193
256	0.701	3.226	0.245	0.368	0.215	0.204
512	5.669	25.604	1.912	0.660	0.349	0.231
1024	45.559	204.471	15.029	1.658	1.153	0.370
2048	368.105	1633.320	120.484	5.749	7.118	1.102

These running times approach what may be realistically expected from a real-time simulator. The sequential timings take significantly longer on the large scale simulations. Comparison between sequential and GPU timings may be better seen in the speedup graphs. The GPU timings include memory transfer from host memory to device memory as it is an important component in the running time of the parallel implementation.

Figure 5(b) shows the speedup achieved by the GPU implementation in a regular scale, to provide perspective on the vast differences in the method types. Since the finer details are difficult to view, Figure 5(c) shows the speedup in a log_2 scale in order to better see comparisons between the speedup achieved in each method. Each of the GPU implementations in this paper ran faster than the CPU implementation on sizes larger than 128x128, with the IMT kernel running faster on all sizes greater than 64x64. Each kernel had the highest speedup value at the 2048x2048 cell size simulation. Unfortunately, due

to running time constraints on the sequential implementation larger scale comparisons would be challenging to provide. The highest speedup achieved was using the Iterative Minimal Time kernel at 229x faster than the CPU implementation. The IMT sequential algorithm is less efficient sequentially than the Minimal Time Algorithm, which explains the disparity in speedup times. The MT kernel had a maximum speedup of 64x, and was the lowest speedup achieved. The atomic operations are the bottleneck in the MT kernel, and further investigation into removing them could improve this kernel. The Burn Distances kernel was significantly faster, 109x, than its sequential counterpart. Overall the results of these tests imply that a GPU-based fire simulator could improve run times significantly and make great strides towards a real-time simulator.

C. Throughput

The graph for throughput may be found in Figure 5(d). The BD kernel performed the best in throughput, showing a steady increase with the increase in input size. The MT kernel performed the worst again, until it passed the performance of the IMT kernel in the 2048x2048 runs. All the kernels dipped in throughput from the 64x64 to the 128x128 tests. Table I shows that the median running times between the two sizes are not vastly different, but the increase in input size decreased the throughput for this size. The dip in performance shows that for these first few kernel runs, the memory transfer to and from the GPU takes most of the processing time.

VII. Conclusion and Future Work

This paper describes the implementation of three fire spread methods which may be used in forest fire simulation. The parallel implementations of the sequential algorithms produced significant speedups in every spread method, ranging from 64x to 229x speedup on the large-scale simulation grid. The results in this paper show the potential for a forest fire simulator to be implemented using the GPU as the processing workhorse. Using the GPU, a real-time simulator can become possible, which would be a vast improvement on the current state-of-the-art. The work presented in this paper is a first step towards a comprehensive forest fire library which could be used to create custom forest fire simulations. A future version for this project incorporates fire acceleration, crowning, and a prototype of spotting were all implemented. This work builds the foundation on which a new simulator may be built. While this work was a prototype system that proved GPU computing was not only possible, but successful at improving runtimes of the fire simulation as compared to a sequential counterpart, the code for this iteration is not available. However, the library code for the future iterations of this project is available through an open-source library [18]. One of the strengths of GPU computing is the ability to have asynchronous data transfer. The ability to transfer partially complete data back to a visualization system would increase the usefulness of a simulator, processing computation while portions of the simulation is being viewed. The kernels will need to be adjusted to account for the asynchronous data transfer.

Acknowledgment

This material is based in part upon work supported by: The National Science Foundation under grant number(s) IIA-1329469, IIA-1301726, and by Cubix Corporation through use of their PCIe slot expansion hardware solutions and HostEngine. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or Cubix Corporation.

References

[1] R. W. Gorte and K. Bracmort, "Forest fire/wildfire protection." Congressional Research Service, Library of Congress, 2006.

[2] R. C. Rothermel and I. Forest, "A mathematical model for predicting fire spread in wildland fuels," *AUSFS*, 1972.

[3] P. L. Andrews, "Behave: fire behavior prediction and fuel modeling system-burn subsystem, part 1," 1986.

[4] E. Pastor, L. Zarate, E. Planas, and J. Arnaldos, "Mathematical models and calculation systems for the study of wildland fire behaviour," *Progress in Energy and Combustion Science*, vol. 29, no. 2, pp. 139–153, 2003.

[5] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents*. Addison-Wesley Professional, 2010.

[6] O. Séro-Guillaume, S. Ramezani, J. Margerit, and D. Calogine, "On large scale forest fires propagation models," *International Journal of Thermal Sciences*, vol. 47, no. 6, pp. 680–694, 2008.

[7] C. D. Bevins, "Firelib user manual and technical reference," *Systems for Environmental Management*, 1996.

[8] M. A. Finney, "Farsite: Fire area simulator: model development and evaluation," *Intermountain Forest and Range Experiment Station, General Technical Report*, 2004.

[9] R. V. Hoang, M. R. Sgambati, T. J. Brown, D. S. Coming, and F. C. H. Jr., "Vfire: Immersive wildfire simulation and visualization," *Computers & Graphics*, vol. 34, no. 6, pp. 655 – 664, 2010.

[10] S. H. Peterson, M. E. Morais, J. M. Carlson, P. E. Dennison, D. A. Roberts, M. A. Moritz, D. R. Weise *et al.*, "Using hfire for spatial modeling of fire in shrublands," 2009.

[11] D. Shreiner, *OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.2*, 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[12] F. Sousa, R. dos Reis, and J. Pereira, "Simulation of surface fire fronts using firelib and {GPUs}," *Environmental Modelling & Software*, vol. 38, no. 0, pp. 167 – 177, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1364815212001867

[13] C. Nvidia, "Programming guide," 2008.

[14] B. Arca, T. Ghisu, and G. A. Trunfio, "Gpu-accelerated multi-objective optimization of fuel treatments for mitigating wildfire hazard," *Journal of Computational Science*, vol. 11, pp. 258–268, 2015.

[15] N. Baranovskiy, "Algorithms for parallelizing a mathematical model of forest fires on supercomputers and theoretical estimates for the efficiency of parallel programs," *Cybernetics and Systems Analysis*, vol. 51, no. 3, pp. 471–480, 2015.

[16] GDAL Development Team, *GDAL - Geospatial Data Abstraction Library, Version x.x.x*, Open Source Geospatial Foundation, 201x. [Online]. Available: http://www.gdal.org

[17] "Cubix corporation, xpander rackmount 8 gen3 16-channel (128gbps) pcie slot expansion system and hostengine." [Online]. Available: http://www.cubix.com/

[18] J. Smith, "vfirelib: A forest fire simulation library implemented on the gpu," Master's thesis, University of Nevada, Reno, 2016.