International Journal of Computer Information Systems and Industrial Management Applications.
ISSN 2150-7988 Volume 5 (2013) pp. 106-114

Fault Tree Analysis Speed-up with GPU Parallel Computing

Hadi Aghassi[1] and Farrokh Aghassi[2]

[1] Electrical and Computer Engineering Depart, Shahid Rajaee University,
Tehran, Iran
hadi.aghassi@gmail.com

[2] School of Engineering, Razi University,
Kermanshah, Iran
farrokhaghassi@gmail.com

***Abstract*: The reliability analysis of critical systems can be performed using fault tree analysis. One of the common approaches used for fault tree analysis is Monte Carlo simulation. The purpose of this paper is therefore to show an algorithm to speed up Monte Carlo simulation for analyzing fault tree with parallel computing in GPU. To this end, we use time-to-failure tree to model fault tree with Compute Unified Device Architecture (CUDA) which is used to accelerate the execution of loops with many repetitions. We also use this technique to accelerate Monte Carlo simulations. In addition, we visualize fault tree so that the user is able to generate fault tree in detail using our developed software and can execute it. The computational outcomes validate the effectiveness of the suggested approach, as we approached about 310 times speed-up in large fault trees.**

Keywords: Fault tree analysis; Monte Carlo simulation; sensitivity analysis; parallel programming; GPU computation; CUDA; time-to-failure

I. Introduction

Fault Trees (FT) [1] have proven to be the most popular choice in terms of building an analytical model of a system. They provide a compact representation of the system that is easily understood by human. Several researchers have proposed different methods to solve fault trees [3]. FTs are analyzed by analytic approaches or Monte Carlo Simulations (MCS) [2].

The analytic approach is fast and computationally cheap. Nevertheless, its usage is limited to a few models and certain kinds of parameter distributions. Moreover, in case the parameters are correlated, this approach proves useless. The combination of *warm* or *cold spares* and *Weibull* (or other non-exponential) *time-to-failure* (TTF) distributions complicates analytic approaches. By increasing dynamic gates in FT, this complication becomes more obvious and this complexity will lead to have more mistakes in FT analysis. Because of this problem, most FT systems are vital and consequently any mistake can go unnoticed and irreparable.

Unlike the analytic approaches, the simulation approach can be broadly used and the failure can be controlled by the number of iterations. However, the simulation approach is limited by its intensive computation. Because of the rare events (such as failures and errors), an extremely large size of sample may be needed for the purpose of obtaining estimates at a high level of confidence. Therefore, MCS is time-consuming [3, 4, 15].

There are two kinds of fault trees including static and dynamic FTs. In analytic approaches, static fault trees can be solved in two ways: (1) using Binary Decision diagram (BDD) [6], (2) using cut sets [5, 6]. The cut set approaches are generally inferior to the newer BDD based approaches. In addition, dynamic FT gates can be solved by conversion to equivalent Markov chain model [6]. To solve FT in analytic approaches most often static and dynamic parts of tree are firstly separated and they are solved afterwards [7]. However, the conversion to equivalent Markov chain in dynamic sub tree is confusing and in a large sub tree, it is very difficult and the risk of mistakes occurring is very high.

In [3, 4, 15], a tree model is presented which can be used for accelerating MCS using field programmable gate arrays (FPGAs). This model can be easily obtained from a FT and has a direct hardware implementation. Because it computes failure time of FT root while a fault occurs in a basic event, it is called time-to-failure (TTF) tree. In fact, a time-to-failure tree receives the TTFs of the components, and computes the TTF of the whole system. In other words, this model is a method for representing the mathematical relation between the system TTF and the components' TTFs. Both dynamic and static FTs can be converted into time-to-failure trees. This model uses FPGAs and works with hardware, therefore it has less flexibility. For example, while FT is changed, the user has to change the TTF and hardware additionally.

Concisely, following methods are used for analyzing FT. We can use BDD to analyze static nodes but it is not useful for dynamic gates and therefore we had to use Markov chain for dynamics. Markov chain is useful for phase-dependent or time-dependent analyses but it is very complex in large FTs. In addition, in most applications, a total probability is all that is desired [8]. Therefore, MCS is a good choice to analyze FT, which has been widely applied in hardware simulation of MCS for FT analysis [3, 4, 15]. In hardware simulation of FT analysis, flexibility is ignored, for it is not practically helpful. Thus, software flexibility makes a case for speedy software for the analysis of FTs.

One of the most eminent aspects of FT is its *coverage factor*, which is the probability that the system can automatically recover from a fault and its associated errors, and therefore it is able to continue its standard operation, even in a degraded mode. We illustrate this subject with more detail in the next section.

Moreover, another aspect of FT analysis is sensitivity analysis. Sensitivity analysis is one of the most important stages in FT analysis because it specifies weakness of a FT. A perfect sensitivity analysis is also time-consuming because all of the events must be evaluated individually or in sets.

In this paper, we use TTF model for a software modeling and show an algorithm to speed up MCS for analyzing fault tree and sensitivity with parallel computing in GPU. For this purpose, we apply time-to-failure tree to model fault tree with CUDA programming language belonged to NVIDIA corporation that is used to accelerate execution of loops with numerous iterations. In addition, we designed a visual software to paint fault tree in such a way that the user can generate the CUDA code and execute it. In this paper, we approached about a 310 times speed-up in FTA and about 100 times in sensitivity analysis.

The rest of this paper is organized as follows: In section II, we investigate the background related works. In section III, our proposed speed-up algorithm in MCS for fault tree analysis and sensitivity analysis is explained in detail, and its implementation and visualization descriptions are discussed in section IV. We evaluated the proposed algorithm in section V comparing it to series algorithms. In the last section, the conclusion and future works are explained.

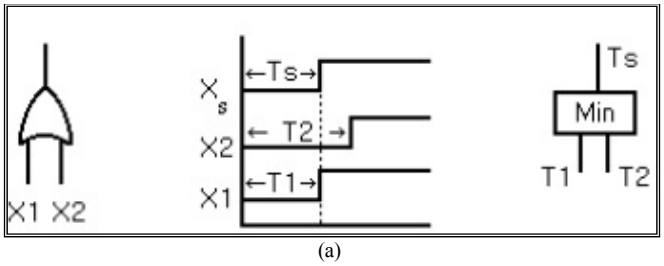
II. RELATED WORKS

Several researchers have proposed different methods to solve fault trees [3, 5, 6]. Dugan et al. [7, 9, 10], have shown that it is possible for the modularization process to identify the independent sub-trees with static or dynamic gates: to use BDD for static sub trees and to use a different Markov model for each dynamic sub tree. Furthermore, A. Ejlali [4] showed a new manner to calculate failure time of FT root with the help of hardware.

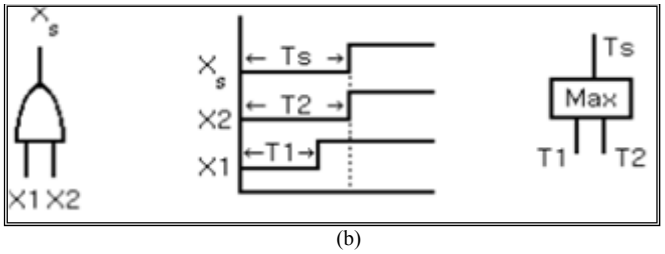
A. Time-To-Failure(TTF)

According to [15] we can easily attribute a TTF component to every FT gate. The fault tree model for a series system is simply an OR gate. In a series system, each element of the system is required to operate correctly for the entire system to operate correctly. Therefore, the TTF of a series system is equal to the minimum of the components' TTFs. "Fig. 1(a)" shows how a MIN unit corresponds to the OR gate in FTs. The fault tree model for a parallel system is simply an AND gate.

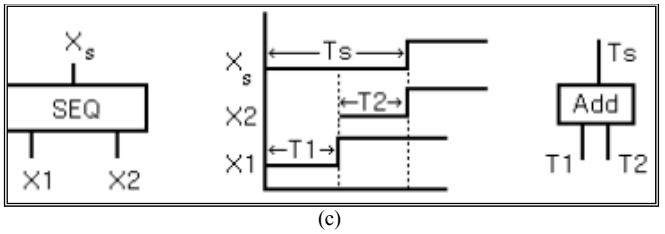
In a parallel system, only one of the several elements must be operational for the system to perform its function correctly. Therefore, the TTF of a parallel system is equal to the maximum of the components' TTFs. "Fig. 1(b)" shows how a MAX unit corresponds to the AND gate in FTs.



(a)



(b)



(c)

Figure 1. The gates in FT and their corespanding in time-to-failure tree: (a) series system (b) parallel system (c) cold spare.

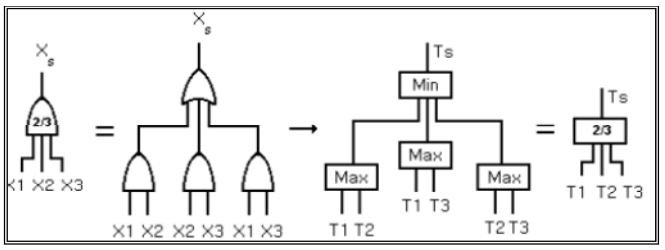


Figure 2. A 2-of-3 gate is converted to its corresponding time-to-failure unit

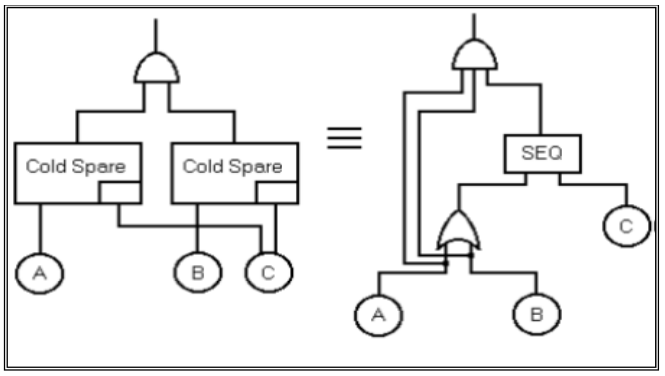


Figure 3. Two cold spares and its FT and TTF

The SEQ gate is one of the dynamic gates, which can be used for modeling cold spares. It forces the input events to occur in the left-to-right order, i.e. an input event to a SEQ gate is not enabled until all of its left inputs have already occurred. "Fig. 1(c)" shows how an ADD unit corresponds to the SEQ gate in FTs.

"Fig. 2" illustrates how an M-of-N gate can be converted to a corresponding time-to-failure tree.

As an example, "Fig. 3" demonstrates how an FT with two cold spare gates can be transformed into another identical FT with a SEQ gate, which in turn can be easily converted to a time-to-failure tree [4].

B. Sensitivity Analysis and Coverage Factor

One of the most important outputs of a FT analysis is sensitivity analysis, also known as importance analysis, which is calculated for the top event. Sensitivity analysis establishes the significance for all events in the fault tree in terms of their contributions to the top event probability. Both intermediate events (gate events) as well as basic events can be prioritized according to their importance. Sensitivity can be calculated as the ratio of sensitivity of the top event

probability to an increase or decrease in the probability of any event in the fault tree. Both absolute and relative sensitivity analyses can be calculated too. In this paper, we discuss the relative sensitivity analysis.

As mentioned earlier, a fault tolerant computer system may fail to recover from a fault, even if spare units remain. For example, a fault may produce an undetected error and then the subsequent calculations or operations may operate on incorrect data possibly leading to overall system failure. Even when an error is detected, the system may still be unable to recover, because the fault could *confuse* the automatic recovery procedures into disabling the wrong component. A coverage model is used to help the structure of our discussion of covered and uncovered faults.

A random-selector unit can be used for modeling the coverage factor in TTF. For example, "Fig. 4" shows how a cold spare with the coverage factor CF=0.7 can be modeled using a random-selector unit. In this figure, T_P is the TTF of the primary unit and T_S is the TTF of the secondary (spare) unit. Here T_{SYS} is equal to T_P with the probability of 0.3 and is equal to T_P+T_S with the probability of 0.7 [4]. A random-selector unit can be easily implemented as a function or a state in software programming.

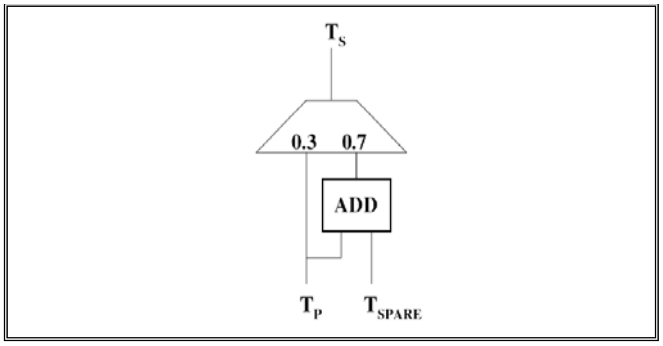


Figure 4. A TTF tree modeling cold spare with coverage factor = 0.7

C. CUDA

In CUDA, the computing system device consists of a host, which is a traditional central processing unit (CPU), such as an Intel architecture microprocessor in personal computers, and one or more devices, which are massively parallel processors equipped with a large number of arithmetic execution units.

In modern software applications, program sections often exhibit a rich amount of data parallelism, a property allowing many arithmetic operations to be safely performed on program data structures simultaneously. The CUDA devices accelerate the execution of these applications by harvesting a large amount of data parallelism [11].

A CUDA program consists of one or more phases that are executed on either the host (CPU) or a device such as a GPU. The phases that exhibit little or no data parallelism are implemented in host code [12]. However, those exhibiting a rich amount of data parallelism must be implemented in the device code.

A CUDA program is a unified source code encompassing both the host and the device code. The NVIDIA C compiler (NVCC) separates the two during the compilation process. The host code is a straight ANSI C code; it is further compiled with the host's standard C compilers and runs as an ordinary CPU process. The device code is written using ANSI C extended with keywords for labeling data-parallel functions, called kernels, and their associated data structures. The device code is typically further compiled by the NVCC and executed on a GPU device [11, 13].

"Fig. 5" illustrates the difference between CPU and GPU. GPU is specialized for compute-intensive, highly parallel computation and is therefore designed such that more transistors are devoted to data processing rather than to data caching and flow control.

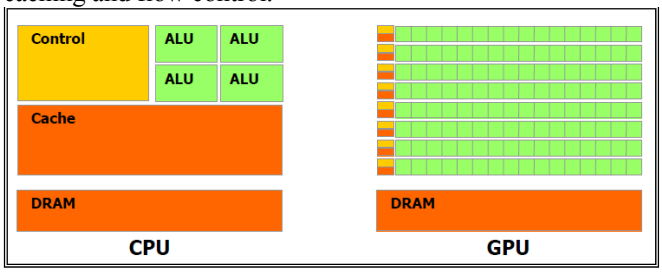


Figure 5. Structure of CPU vs. GPU

When we program through CUDA, the GPU is viewed as a computing device capable of executing a numerous number of threads in parallel. It operates as a coprocessor to the main CPU. In other words, data-parallel, compute-intensive portions of applications running on the host are off-loaded onto the device.

More precisely, a portion of an application that is executed many times independently with the same algorithm and on different data, can be isolated into a function that is executed on the device as many different threads. To that effect, such a function is compiled to the instruction set of the device and the resulting program, called a *kernel*, is downloaded to the device to be executed [11, 13].

Both the host and the device maintain their own DRAM, referred to as *host memory* and *device memory* respectively. One can copy data from one DRAM to the other through optimized API calls that utilize the device's high-performance Direct Memory Access (DMA) engines.

The amount of performance benefit an application will realize by running on CUDA depends entirely on the extent to which it can be parallelized. The code that cannot be sufficiently parallelized should run on the host, unless doing so would result in excessive data transfers between host and device [13].

Amdahl's law [14] specifies the maximum speed-up that can be expected by parallelizing portions of a serial program. Essentially, it states that the maximum speed-up (S) of a program is:

$$S = \frac{1}{(1-P)+\frac{P}{N}} \quad (1)$$

Where P is the fraction of the total serial execution time taken by the portion of code that can be parallelized and N is the number of processors over which the parallel portion of the code runs.

The larger N is (that is, the greater the number of processors), the smaller the P/N fraction. It can be simpler to view N as a very large number, which essentially transforms the equation into $S = 1/(1-P)$. Now, if ¾ of a program is parallelized, therefore the maximum speed-up over serial code is $1 / (1 - ¾) = 4$.

For most purposes, the key point is that the greater P is, the greater the speed-up [4]. An additional caveat is implicit in this equation, which is that if P is a small number (so not substantially parallel), increasing N does little to improve performance. To get the largest lift, the best practices suggest spending most effort on increasing P; that is, by maximizing the amount of the code that can be parallelized.

When using CPU timers, it is critical to remember that many CUDA API functions are asynchronous, that is, they return control back to the calling CPU thread prior to completing their work. All kernel launches are asynchronous, and so are all memory copy functions with the Async suffix on the name. Therefore, to accurately measure the elapsed time for a particular call or a sequence of CUDA calls, it is necessary to synchronize the CPU thread with the GPU by calling cudaThreadSynchronize() immediately before starting and stopping the CPU timer. cudaThreadSynchronize() blocks the calling CPU thread until all CUDA calls previously issued by the thread are completed [11]. These methods are available in the CUDA library.

III. PROPOSED ALGORITHM

As mentioned earlier, for analyzing FT we have two ways: analytic approaches or MCS [2]. Analytic approaches are speedy and liable to mistake, and their calculations are limited to some special kinds of FTs, but MCS is time consuming, unmistakable, and is not limited in FT kinds [4]. Analytic approaches are inefficient for calculating large FTs and mostly MSC is used for this purpose, though MCS is so time consuming. Considering this fact, in this paper, we propose a method with parallel computing in GPU to decrease the execution time of the MCS that includes numerous iterations.

However, in recent studies on FT, not only fault tree analysis includes the computation of failure time of FT that is described in the first phase, but also it takes account of sensitivity analysis for better understanding of FT. This analysis is also time-consuming and therefore we offer a parallel solution for this problem in the second phase.

A. *Speed-up algorithm to Monte Carlo simulation in FT analysis*

In MCS, we have to generate random numbers for FT entries and iterate this task for many times. The generated random numbers must be exponential or Weibull distributions [4]. In this paper, we choose exponential distribution to analyze FT. For this purpose, if variable R has a uniform distribution over [0,1), the random variable X is an exponential distribution of [4]:

$$X = -\frac{\ln(1-R)}{\lambda} \qquad (2)$$

Where variable λ is the failure rate of each component and its scale is mostly hour. Generated number X is the time to failure of the same component with λ [4].

Each leaf of FT has a λ. For all MCS iterations, we set a λ to each FT leaf, and generate X for that leaf. Then, we pass X to the top nodes of FT with MAX, MIN, ADD, or other gates. We keep running this operation until we arrive at the root of FT. Consequently, each iteration computes an X that contains failure rate of FT. Ultimately, Xs average of FT root is calculated, which is known as the FT average failure rate.

In the algorithm of "Fig. 6", first, we assign the root of FT to the function argument E and then calculate X for each leaf of fault tree. Note that calculate times of X in this algorithm are equal to the number of MCS iterations for each leaf. Then this array is propagated to the top of FT. As an alternative, we could also execute the entire algorithm in MCS iteration count as in each iteration, we could allocate one X to each leaf, and propagate it to the top of FT. The second option seems to be useless in this project because it needs more iterations with heavier codes to be executed in each iteration.

In the first iteration of the *for loop*, we firstly define a two dimensional array to save X of leafs. The first dimension is leaf ID and the second is iteration number. Then, for each iteration, we assign an X to this array.

```
Node E = Root of FT
Array Prob[][] =
        new[leaf counts][iteration of MCS]

For(i = 0 to tree leaf count)
{
        For(j = 0 to iteration of MCS)
        {
                Calculate X to every leaf;
                Prob[i][j]=X;
        }
}
double FTAnalysis (Node E)
{
        If E is Leaf then
                Return Prob[E];

        Else if E is an AND node then
        {
                FTAnalysis(E child);
                Calculate max Prob of children
                        with CUDA;
                Insert max Prob in E node;
                Return max Prob;
        }
        Else if E is an OR node then
        {
                FTAnalysis(E child);
                Calculate min Prob of children
                        with CUDA;
                Insert min Prob in E node;
                Return min Prob;
        }
        Else if E is a sequence node then
        {
                FTAnalysis(E child);
                Calculate sum Prob of children
                        with CUDA;
                Insert sum Prob in E node;
                Return sum Prob;
        }
        Else if E is a cold, warm or hot  node then
        {
                Convert this node to equivalent
                        TTF that will be a tree ;
                Calculate FTAnalysis(E);
        }
        Else if E is a FDEP  node then
        {
                Convert this node to equivalent
                        TTF that will be a tree ;
                Calculate FTAnalysis(E);
        }

        Else  // E is a intermediate node
                Return FTAnalysis (child of E);
}
```

Figure 6. FT algorithm

In the next step, recursive FTAnalysis() function is called, which firstly checks the node to see whether it is a leaf or

not. In case of a leaf, Prob (probability) variable of the node is returned. Otherwise, if the type of the node is an AND one, then children of the node are checked, and if they are not leaf then FTAnalysis() function for children is called to determine Prob of the nodes. After the determination of children's Probs, another function is called to calculate the maximum of children's Probs in GPU. This function is a CUDA program that obtains and returns children's Probs. Now, node Prob is equal to the max of children's Probs. Considering the fact that iteration count is high and Prob variable dimensions count is equal to iterations count, calculating this amount of information is time-consuming and is considered to be processed in GPU. Subsequently, we attribute the returned value from GPU to node E and then return it to be used in other nodes.

If node E is an OR or SEQ gate, the procedure will resemble an AND gate. However, in CUDA function of the OR gate, we must calculate the minimum of children, and in SEQ gate, we must calculate the sum of children's Probs.

If E node is a cold, warm, or hot spare or a FDEP gate, then we convert these gates into equivalent time to failure tree to calculate FT. Afterward, converted tree is analyzed by FTAnalisis() function. In dynamic states, we could also define a program easily to calculate the failure probability of these nodes, because of the advantage of software programming flexibility. Nevertheless, we use converting gates to TTF to collaborate with TTF idea and to compare this algorithm to TTF model. To this end, first we have to change existing dynamic fault tree nodes to equivalent TTF model and add some codes to FT algorithm expressed in "Fig. 6". For instance, the TTF model for cold spare is "Fig. 4" and its algorithm can be like "Fig. 7".

```
int j=a random integer between 0 and spare child count-1
For(int i 0 to spare child count)
        If  i=j
                return FTAnalysis(child i);
```

Figure 7. random-selector algorithm

In this algorithm, we used CUDA functions to accelerate the execution of the program. To define these functions, some prerequisite tasks are necessary to be accomplished at first. One of these tasks is to determine dimensions of CUDA function. These dimensions are used to define the number of grids, blocks, and threads in a CUDA program. For example, "Fig. 8" defines 10*10 threads in a 4*4 blocks in a GPU.

```
Dim3 blocks = (4, 4);
Dim3 threads = (10, 10);
```

Figure 8. Defining threads and blocks

The blocks variable defines the number of blocks that has a two-dimensional index and for each block, we have two dimensional indices of threads, so in this example we have 10*10*4*4=1600 threads [11, 13]. All of these dimensions are three-dimensional variables and if any of them were not defined, they would be assumed as one-dimensional. Therefore, in this example, block dimensions are (4, 4, 1) and thread dimensions are (10, 10, 1).

For calling CUDA code, we must do like "Fig. 9":

```
Ret-type func-name<<<blocks, threads>>>(arguments);
```

Figure 9. Calling a CUDA function

This code executes a desired function for each thread of a block. In other words, the code will be executed for blocks*thread times. Since GPU has numerous cores, each thread is executed in one core, consequently the speed of the execution rises. Therefore, for more speed-ups, more cores are a good choice.

B. Speed-up algorithm to sensitivity analysis

For a FT analysis, determining failure time of fault tree root is not adequate, because this just identifies the uncertainty analysis of fault tree, and does not make any suggestions about the answer to the following questions: "where is the problem or the most faulty part in FT?" or "how do results change by changing input parameters?" or "what is the most cost-effective way to improve reliability?". Sensitivity analysis provides some information about the important nodes that have the most effect on the root of FT.

Sensitivity analysis is one of the major outputs of an FTA, which is calculated for the top event. This top sensitivity analysis establishes the significance for all of the events in the fault tree in terms of their contributions to the top event probability [8].

One useful point about sensitivity analysis is that they generally show relatively few events contributed to the top event probability. In many of the past FTAs, less than 20% of the basic events were important contributors in the fault tree contributing more than 90% of the top event probability. Moreover, the importance of the events in the fault tree generally clusters into groups that differ by orders of magnitude from one another. In these cases, the importance is so dramatically diverse that they are not generally dependent on the preciseness of the data used in the FTA [8].

As mentioned above, there are two kinds of sensitivity analyses: absolute and relative sensitivity analysis. Absolute sensitivity analysis is calculated with partial derivative of basic events that are not very time-consuming. On the other hand, relative sensitivity analysis is calculated by changing basic events and analyzing top event changes. In this paper, we discuss relative sensitivity analysis, and call it sensitivity analysis.

In a *sensitivity analysis*, an input data parameter, such as a component failure rate, is changed, and the resulting change in the top event probability is determined. This is repeated for a set of changes either using different values for the same parameter or changing different parameters, for example, changing different failure rates. Usually for a given sensitivity evaluation, only one parameter is changed at a time. This is called a one-at-a-time sensitivity study. Sometimes two or more parameter values are simultaneously changed to study the interactions among the parameters.

The decisions that are made to carry out sensitivity analysis include which data parameters to be changed and what values to be used for the parameters. A small change in a data parameter shows the linear effect of the change on the top event probability. Assigning 1 or 0 or a value to the failure probability of a basic event provides the maximum effect of changes in the parameter. It can be shown that using a small change or using 1 or 0 for the event probability gives the same information as the sensitivity analysis [8]. Therefore, intermediate changes in the parameters are selected for the sensitivity studies.

As indicated in algorithm "Fig. 6", we defined a Prob variable to save failure time of FT, and in this case, we can use this variable and the same algorithm to calculate FT failure time. Instead of MCS iterations, we implement some other iterations in which one of the probabilities of leaves changes. In other words, first loop of the algorithm presented in "Fig. 6" resembles the algorithm in "Fig. 10".

```
Array Prob[][] =  new[leaf counts+1][leaf counts];
For(j = 0 to leaf count)
{
        Calculate X for every leaf;
        For(int i = 0 to leaf counts +1)
                Prob[i][j]=X;
}
For(i = 1 to tree leaf count+1)
        Prob[i][i-1]= Prob[0][i-1] + 0.001;
```

Figure 10. Section of sensitivity analysis algorithm

In this algorithm, first we define a two-dimensional array whose first dimension is "the number of leaves + 1" and the second is "the number of leaves". This two-dimension matrix saves some random numbers calculated by the X formula in all lines. These numbers are default numbers and then in each line of matrix, one of them is changed. The second loop calculates the rest of the matrix by summing previous random number and 0.001 to see changes in the fault tree root. The rest of the algorithm is the same as the algorithm in "Fig. 6" and ultimately, all of the root sensitivities for basic event changes are inserted in Prob variable of root node to be checked and analyzed.

In this algorithm, we could add some policies to indicate sensitivity answers in the end of the algorithm. To this end, we could sort fault tree analysis answers and show the greatest answer, that is the node related to this answer has the greatest importance in FT. Furthermore, safety of this node should be considered.

We utilized only one node change in the algorithm, however, sometimes it is necessary to change two or more nodes concurrently to see FT sensitivity. For this purpose, we could obtain the leaves that should have been changed by the user, and we could change them, and execute FTAnalysis() function to see the answers. In addition, we could change all of the nodes in pairs or in sets respectively, and send them to the function to get answer.

If the fault tree used is a big one, the presented sensitivity algorithm will be very functional, because sensitivity analysis is a manual analysis and in a massive analysis, it is likely to make a mistake. Therefore, this algorithm is very useful in large fault trees.

Sensitivity analysis is almost a troublesome one and the majority of activities to calculate sensitivity analysis are manual methods. Besides, in large FTs, determining nodes to be changed is very complicated task. On the other hand, there is not a distinctive technique to analyze the sensitivity of a fault tree; hence, the presented sensitivity analysis algorithm in "Fig. 10" is not a consistent algorithm. In other words, since sensitivity analysis is very changeable, it is likely to change this algorithm to another form according to the objective.

On the whole, for analyzing fault tree, we speed up MCS and sensitivity analysis, and since these two are adequate to analyze a fault tree in action, thus we only offer solutions that are almost enough for analyzing every fault tree. In particular, the proposed method is very helpful in analyzing large FTs. In addition, it reduces mistakes that are very frequent in other methods. For this reason, the proposed method has obvious advantages over other earlier works on analyzing large FTs.

IV. IMPLEMENTATION

For better visualization, we designed a user interface, which is illustrated in "Fig. 11" and is developed with Java programming language by NetBeans IDE (7.0). This application contains some modules of FT design for user's convenience. In this UI, the user is able to design his own FT by gates, events, or other equipments that are included in a board on the application design tools.

In addition, there is a fault tree designing area that has a dynamic scroll bar. To illustrate this aspect of application, suppose the FT designed by user is so large that cannot be placed in the design area. In this case, user can zoom out the design area, move to an unoccupied area, and then zoom in to keep drawing the rest of tree. To take this fact into consideration, we must say that the reason is in consistent with scroll bar, which means that in zooming in or out, scroll bar has a constant length.

This UI gets the FT designed model from the user and then saves it in an XML file. When we save the FT in an XML file, the UI checks the FT, and if there is any mistake or nonstandard gates, then the application can correct and save it. In a regular FT analysis, we should save dynamic gates too, however, sometimes it is necessary to change dynamic gates to static ones and save them. In this case, UI checks the FT, and if there is a cold, warm, or hot spare and FDEP, then UI changes it to the static gates (OR, AND or SEQ) [3, 4, 15]. In this manner, the generated TTF will be very simple to use, and executed time will be less because dynamic nodes are very complicated and time-consuming in execution.

This application also supports encapsulation concepts because when user clicks in the analyzer of FT button to see the FT analysis answer after execution, he/she will see the FT failure rate. In this case, the user will not be involved in the CUDA code and it will be executed automatically.

Then, by means of the remote procedure call method, the UI calls the program written in C programming language including the CUDA code. Calling this code causes to read the FT saved in XML file and to execute the code. At last, remote called method will return the FT failure rate and the UI will get and show it.

This UI also has a button to arrange the FT designed by user. This button will arrange the entire designed tree to a symmetrical tree that is convenient to understand. Additionally, the user can calculate the FT probability with probabilities attached to each leaf (the probability for each node also should be identified by the user). In addition to analyzing fault tree with Monte Carlo simulation, the user has a choice to calculate his fault tree manually.

Moreover, the user can save FTs in an XML file and later he or she can load and change it. We use XML files to save because of their power. In fact, XML files are standard, machine friendly, and all of the operating systems support them, and they can be used on the internet and all programming languages.

In the programming section, it is necessary to note that CUDA programming language just supports variables of data types defined by C language and other data types, such as objects defined by the user, must be in host memory, and their used variables should pass to device memory. If array data type is defined by a user-defined object, it cannot be passed to devise memory.

To execute CUDA code, we used a NVIDIA GeForce 8600 GS graphic card that has 16 CUDA processors of 600 MHz, core clock rate. The CPU used for this examination was Intel core 2 Duo with 1.83 GHs clock rate. Moreover, the RAM was 2 GB with 987 MHs clock rate. This implementation of the code was done using a DELL laptop with a XP/32 bit windows.

We implemented this algorithm for CPU and GPU in the same manner to compare the results obtained from each of them. The CPU implementation was in C programming language and GPU implementation was in both C and CUDA programming languages, C for host code and CUDA for devise code. We compared two FTs, first with 5 static nodes and then with 20 static nodes. For each FT, we calculated the execution time with the iterations 1, 10, 100, 1000, 10000, 100000 and 1000000. The execution times for GPU are shown with blue color and execution times for CPU are showed with red color.

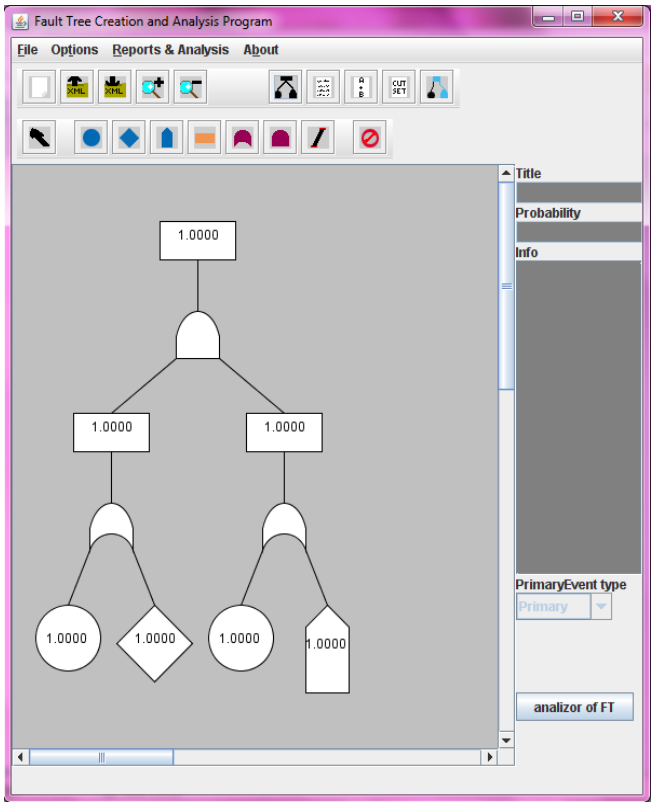


Figure 11. FT design user interface

V. EVALUATION

As mentioned earlier, we implemented our speed-up algorithm with two approaches: CPU FT computation and both CPU- GPU FT computation.

"Fig. 12" is a FT with five nodes and it shows that the GPU calculation of FT begins from about 100 ms when iterations count is 10, and finally in the 1000000 iterations, it reaches about 1s. However, the CPU calculation of FT when iterations count is 10 begins in 1 ms and then grows up exponentially where in 1000000 iterations it reaches 10000 second. This figure shows that the increasing of the GPU computation is almost linier but the CPU computation is exponential. Therefore, in large iterations the CUDA works better than C programming language. Also, this figure shows that after 100000 iterations, CUDA programming language will work better than C programming language and will fasten the FT computation.

"Fig. 13" is FT with 20 nodes and like "Fig. 12" it shows that GPU calculation of FT begins from about 100 ms when the iterations count is 10 and grows up almost like a linier function, but the CPU calculation of FT, when the number of iterations is 10, begins in 1 ms and then grows up exponentially where in 1000000 iterations it reaches 10000 seconds. Also, this figure shows that after 30000 iterations, the CUDA programming language will work better than C programming language.

These two diagrams show that with the development of FT, its CPU computation will increase, but in GPU computation, this increase is not too much, therefore in addition to working better in the large iterations, this algorithm works better in large FTs. In case FT is large and is for a critical system, we have to increase the iteration count so many times to reach a precise FT computation answer. For such a system, this algorithm works very well and it will decrease the time that is necessary for computing FT failure rate.

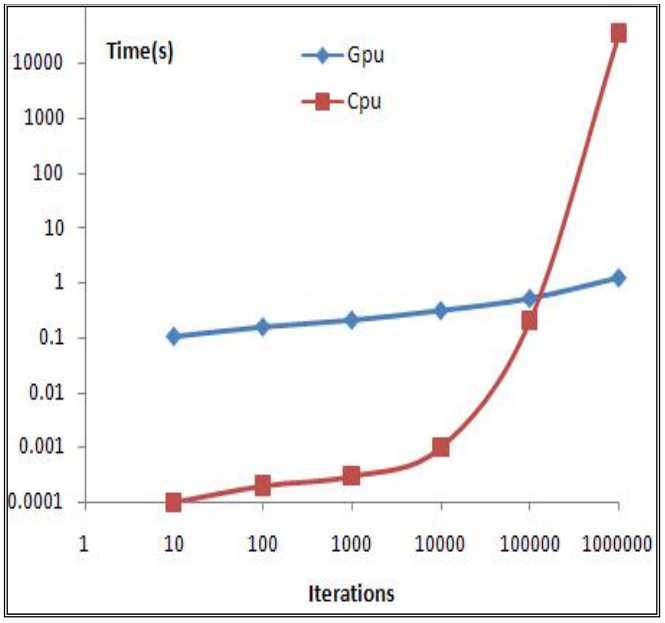


Figure 12. FT calculation with 5 nodes

TABLE I. EXPERIMENTAL RESULTS

FT	Speed-up
For Fault Tree existing of 5 nodes	10
For Fault Tree existing of 20 nodes	24

The experiments show that the implementation of the algorithm can significantly reduce the required time for MCS, especially when the fault tree is large. In the normal FTs (not very large or very little), we have a speed-up about 310 times as long as that of CPU when FT nodes were 100 for a Fault Tree with 100000 iterations, we had 310 times speed-up.

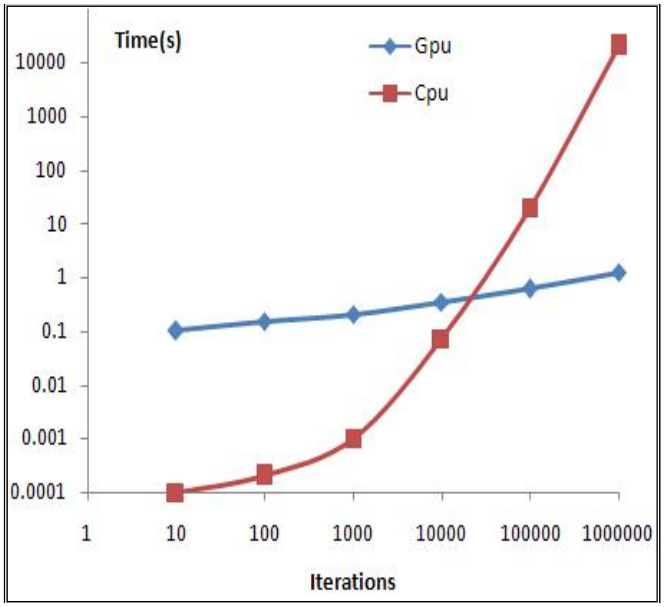


Figure 13. FT calculation with 20 nodes

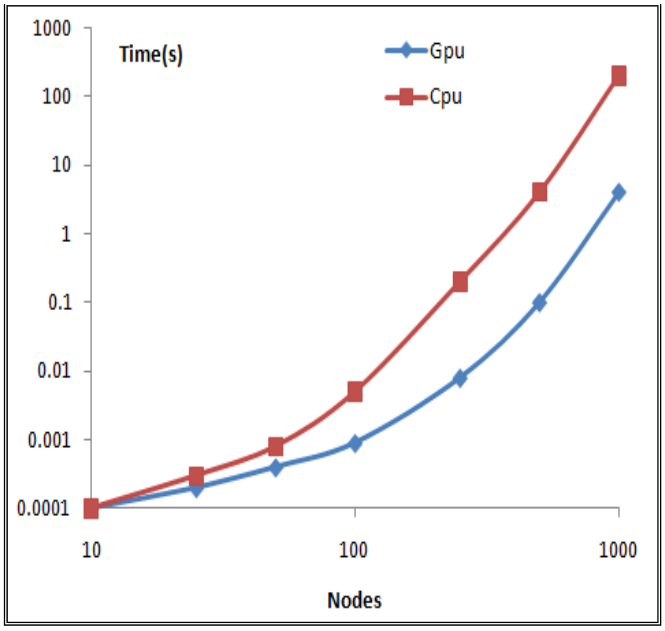


Figure 14. Sensitivity analysis of FT in CPU and GPU

"Fig. 14" is a diagram to compare sensitivity analysis speed in CPU and GPU processors. In this diagram, we can see that when node counts is 10, then sensitivity analyzing time for both GPU and CPU is approximately 0.0001. When node count is 100, then GPU is 10 times speedier than CPU. Moreover, when node count is 1000, speed-up rises to 100 times. Therefore, in sensitivity analysis we approached a speed-up of about 100 times in proportion to similar algorithm in CPU.

VI. Conclusion

Fault tree analysis can perform the reliability analysis of critical systems. One of the approaches for the fault tree analysis is Monte Carlo simulation. In this paper, we purposed a novel approach to speed up Monte Carlo simulation for analyzing fault tree and to sensitivity analysis with parallel computing in GPU. Using the time-to-failure tree, we modeled fault tree with Compute Unified Device Architecture (CUDA), which is used to accelerate the execution of loops with many repetitions.

Moreover, we accelerated Monte Carlo simulations with our new approach. In addition, we developed a user-friendly software to visualize fault tree so that the users can generate fault tree in detail.

The computational outcomes validate the effectiveness of the suggested approach, as we approached about 310 times speed-up. Our software can obtain much more than 310 times speed-up by using stronger GPU. Our future work is the implementation of a CUDA code generator for the people who do not know CUDA so that they can write their codes in other languages and this generator can exchange their codes into CUDA.

References

[1] E. J. Henley, and H.Kumamoto, "Probabilistic Risk Assessment, Reliability Engineering, Design, and Analysis", *IEEE Press*, 1992.

[2] Metropolis N., and S. Ulam, "The Monte Carlo Method", Journal of American Statistical Association, Vol. 44, No. 247, Sept. 1949, pp. 335-341.

[3] Farrokh Aghassi, Hadi Aghassi, and Zahra Sheykhlar, "A speed-up algorithm in Monte Carlo simulation for fault tree analysis with GPU computing", International Conference of Soft Computing and Pattern Recognition (SoCPaR), 2011, pp. 469-474.

[4] A. Ejlali, S.G. Miremadi, "Time-to-failure tree," Proc. of Ann. Reliability & Maintainability Symp., 2003, pp. 148-152

[5] W. Lee, D. Grosh, F. Tillman, CH. Lie, "Fault tree analysis methods and applications: a review," IEEE Trans Reliab; vol. 34, 1985, pp. 194-302

[6] W. Vesely, F. Goldberg, N. Roberts, D. Haasl, "Fault tree handbook," United States Nuclear Regulatory Commission, January 1981.

[7] J.B. Dugan, R. Gulati, "A modular approach for analyzing static and dynamic fault trees," In: Proc Ann Reliability & Maintainability Symp, Philadelphia, Pennsylvania, USA, January 1997, pp. 57-63.

[8] M. Stamatelatos, W. Vesely, "Fault Tree Handbook with Aerospace Applications", Prepared for NASA Office of Safety and Mission Assurance, 2002, page 89-90-94.

[9] J.B. Dugan, S. Bavuso, M. Boyd, "Dynamic fault-tree for fault-tolerant computer systems, " IEEE Trans Reliab, vol. 41, no. 3, 1992, pp. 363-76.

[10] H. Zhu, S. Zhou, J.B. Dugan, K.J. Sullivan, "A benchmark for quantitative fault tree reliability analysis," Proc. Ann. Reliability & Maintainability Symp., Jan 2001, pp. 86-93.

[11] NVIDIA Corporation, "CUDA programming guide", version 1.1, 2007.

[12] D. Kirk, H. Wen mei, "Programming massively parallel processors," ELSEVIER Inc, 2010.

[13] J. Sanders, E. Kandrot, "CUDA by example: an introduction to general-purpose GPU programming", NVIDIA Corporation, version 1, 2011.

[14] NVIDIA Corporation, "NVIDIA CUDA C Programming, Best Practices Guide", 2009.

[15] A. Ejlali, S.G. Miremadi, "FPGA-based monte carlo simulation for fault tree analysis," Microelectronics Reliability, vol. 44, 2004, pp. 1017-1028

Author Biographies

Hadi Aghassi teaches in the Computer Science Department of Shahid Rajaee Teacher Training University, SRTTU, since 2010. He is Master in Computer Science, Iran University of Science and Technology in 2009, and has a BS degree in Computer Engineering, 2006. He teaches since 2007. His research interests include machine learning, web/text mining, project management, virtual robots, and parallel algorithms.

Farrokh Aghassi received the BS degree in Computer Engineering from Razi University in 2011. He worked in an IT consulting firm and an Investment Bank as project manager. His research interests include parallel algorithms, GPU programming, multiple classifier systems, and neural networks.