Parallel Algorithm for Generation of Test Recommended Path using CUDA

Zhao Yu[#1], Jae-Han Cho[#2], Byoung-Woo Oh*[3], Lee-Sub Lee*[4]

[#]Master's Course, Department of Computer Engineering, Kumoh National Institute of Technology, Korea
[*]Associate Professor, Department of Computer Engineering, Kumoh National Institute of Technology, Korea
[1]zhaoyuivanzhao@yahoo.com.cn
[2]jaehanfs@gmail.com
[3]bwoh@kumoh.ac.kr
[4]eesub@kumoh.ac.kr(Corresponding Author)

Abstract—Software testing of an application makes the user to find defect. The users, called testers, should test the various situations with test cases. In order to make test cases, many states and events have to be considered. It takes much time to create test cases with many states and events. Instead of using the common sequential algorithm, this paper proposes a parallel algorithm for generation of test cases. The proposed method achieves efficient performance using General-Purpose GPU (GPGPU), especially CUDA.

Keyword-Software Testing, Test cases, Parallel algorithm, GPU, CUDA

I. INTRODUCTION

Software testing is an important process in the software system development cycle. If a software has defect, there are many problems to use the software. In order to solve these problems and make the software listed, it doesn't only need to make great effort and spend much time to implement the software, but it also needs to test the software constantly. However, a lot of time is needed to test the software that has a very complex structure, such as the large commercial application software.

In order to overcome these shortcomings, this paper makes use of parallel technology based on CUDA (Compute Unified Device Architecture). CUDA is developed by Nvidia. Many research uses CUDA to enhance performance [2, 4, 5, 9, 10, 11]. Graphic Processing Unit (GPU) has been improved to process general purpose parallel processing [2]. General-Purpose GPU (GPGPU) can process problems of software engineering [6, 7, 8]. Considering the complexity of test case, CUDA can be applicable for generating test recommend path automatically.

In this paper, an algorithm is proposed for generation of test recommended path using instruction set of CUDA and GPU memory. By using this algorithm, new test cases can be generated from the existing test cases. By utilizing parallel technology, the performance of the algorithm has a considerable improvement.

This paper is divided as follows. Section II describes the related work and information about previous work. Section III shows the design details of the algorithm. Section IV presents experimental result. Finally, section V presents a conclusion and future work.

II. RELATED WORK

CUDA makes developers access to the virtual instruction set and memory of the parallel computational elements in GPUs. Using CUDA, the Nvidia GPUs become accessible for computation like CPUs. Unlike CPUs however, GPUs have a parallel throughput architecture that emphasizes executing many concurrent threads slowly, rather than executing a single thread very quickly. This approach of solving general purpose problems on GPUs is known as GPGPU.

The Reverse Engineering Based Automatic Generation of GUI Test Case introduces an algorithm implemented in Python by using which recommended test cases are generated from exiting test cases automatically [1]. This is a simple example to explain the expanded test case generation process.

```
Test case number = 3;
Test case length = 2;
Event number = 4;
State number = 5;
Test cases: {
{[E1, S1], [E3, S5]};       //test case1
        {[E2, S2], [E4, S3]};       //test case2
```

{[E2, S2], [E3, S4]}; //test case3
}

Fig. 1 shows that test case2 and test case3 have the same transition [E2, S2], so in the TFG tree there is only one node. Then collect information from this TFG and manipulate data using GPU. When obtaining the expanded test cases, the algorithm adds them to the TFG. Fig. 6 shows the expanded TFG.

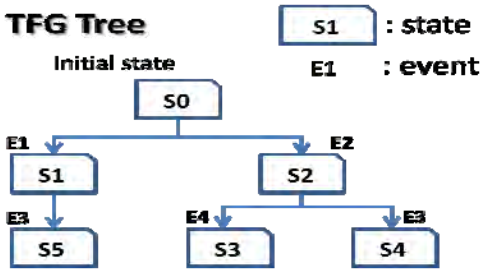

Fig. 1. Test flow graph

In Fig. 2, the dotted line links every node with its expanded nodes. The initial node S0 only has E1 and E2 two events, so E3 and E4 will be generated as the expanded data. For state S1 and S2, they have a similar situation to state S0. For the leaf nodes S3, S4, S5, they have no child node, so the 4 events all need to be expanded.

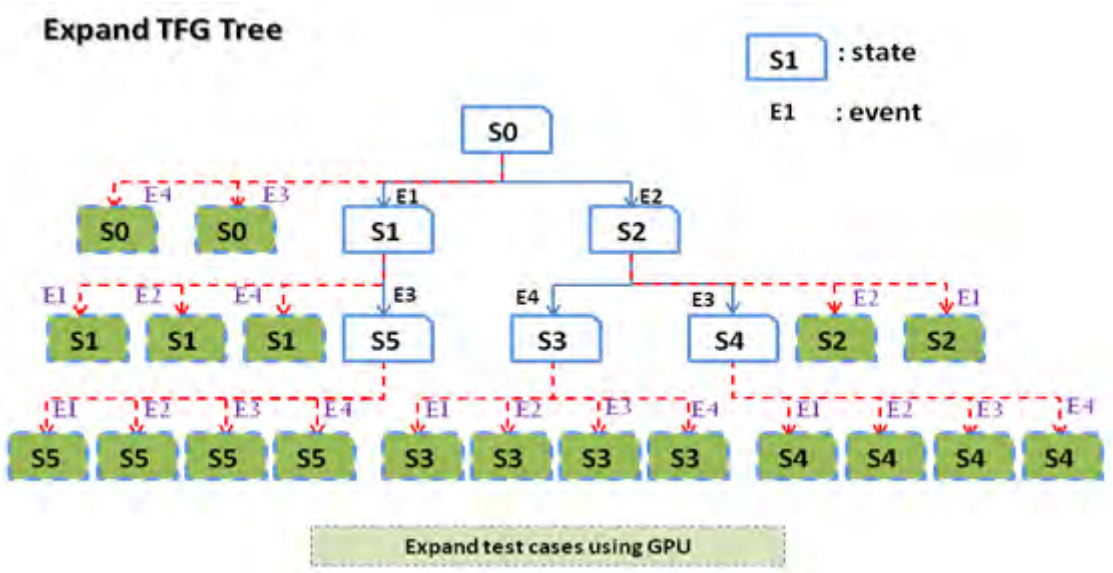

Fig. 2. Expanded TFG

III. Proposed Algorithm

This section shows the main idea of the algorithm and the core part of CUDA algorithm. In order to utilize the powerful computational capabilities of the GPU, the computation part of the algorithm is completed by CUDA. The other parts of the algorithm are pulled off by the CPU. The rest of this section will present the design idea, the explanations of core functions and the detail of the CUDA algorithm.

In this paper, we propose a new algorithm implemented in C language using CUDA. In addition, in order to make a comparison between sequential and parallel implementations, we also implement the algorithm only in C language. Fig. 3 shows the process of the algorithm proposed in [1].

Step 1: TFG generation;
Step 2: Gather event information;
Step 3: Expand test cases;
Step 4: Execute expanded test cases by using **Sikuli Tool**;
Step 5: Analyze and improve;
Step 6: Execute improved test cases;

Fig. 3. The basic process

A. Design Idea

The main idea of the algorithm is that the complex computation parts of the algorithm will be completed by the GPU instead of the CPU. As Fig. 4 shows below, at first the algorithm will be initialized, and then the data will be analyzed and the TFG (the TFG will be explained in the part B) will be built, after that the algorithm will extract information from the TFG, when the whole information is gathered, the GPU will begin to compute, at last, the algorithm gets the result from GPU and add the result to the TFG.

```
1. getEvent();                  //collect all events appearing in given test cases
2. generateTFG();               //generate TFG (Test Flow Graph) using exiting test cases
3. collectDataForGpu();         //extract useful information from the TFG
4. __global__ expandTestCase(); //expand test cases using CUDA
5. addNodes();                  //add the expanded data to the TFG
```

Fig. 4. Pseudo code of the algorithm

B. Generating TFG and Collecting Information

For every test case, its default length is 10, so it has 10 transitions. The structure of the TFG is like a tree. Every path from the root node to a leaf node of the TFG is a test case.

For every node of the TFG, this paper uses “struct” in C language to present it. Fig. 5 shows the “treecase struct”. The “int value[2]” is used to record the current event and state of the node. The “treecase *cnode” is used to link the next nodes that are transitions appearing in the test cases. The “treecase *expendcase” is used to link the expanded nodes. So when the TFG is generated, the *expendcase is still empty.

```
typedef struct treecase{

    int value[2];

    treecase
```

Fig. 5. The memory structure for TFG nodes

After generating the TFG, the number of the nodes of the TFG could be obtained, and then extracting information process will execute from every node of the TFG. The information of every node includes the number of the next-level nodes and the event of the every next-level node. For example, if a node has 3 children and the event of every child is e1, e2 and e3 respectively, the information extracted of this node is {3, e1, e2, e3}. There are many nodes in the TFG, so a two-dimensional array is used to collect all information.

C. Expanding Test Case Using CUDA

After getting the number of nodes of the TFG and information of every node, the algorithm will expand test cases using GPU. Every thread in GPU corresponds to one node of the TFG. So every thread generates expanded test cases only for one node. This parallel way is much faster and more efficient than sequential version. However, using GPU must send data from host to device and receive result data from device to host, if the data volume is too big, it will spend much time in delivering information. In order to avoid this situation, a one-dimensional array is used to store the result data. This paper combines the number of the expanded nodes and the event of the every expanded node to be one data. For example, if a node has 3 expanded nodes and the event of every node is e1, e2 and e3 respectively, the data is {e1e2e13}. It’s just one data. Using this kind of structure array, the time can be reduced obviously. Fig. 6 shows the core part of expandTestCase() function.

D. Adding Expanded Test Cases to the TFG

After getting the expanded data using GPU, the algorithm will add the expanded nodes to the TFG generated previously.

The algorithm is designed recursively. It traverses the whole TFG using depth-first search method. For every node, if there are expanded nodes, add all expanded nodes to the node. At last, the TFG tree becomes a complete tree including original test cases and expanded ones.

```
IF (tid*blockNum+bid) < nodeNum THEN
  WHILE evlist[i]!=-1 DO
    IF count==EVENTNUM THEN
      break;
    ENDIF
    FOR j=0 to *(data + ((EVENTNUM + 1) * (tid * blockNum +
    bid))) DO
      IF evlist[i] == *(data + ((EVENTNUM + 1) * (tid *
      blockNum + bid)) + j + 1) THEN
        break;
      ENDIF
    ENDFOR
    IF j == *(data + ((EVENTNUM + 1) * (tid * blockNum +
    bid))) THEN
      int evCount = *(result + (tid * blockNum + bid));
      int currentEvent = evlist[i];
      evCount++;
      FOR k=0 to evCount%10 DO
        currentEvent*=10;
      ENDFOR
      evCount = evCount + currentEvent;
      *(result+(tid*blockNum+bid)) = evCount;
    ENDIF
    i++;
    count++;
  ENDWHILE
ENDIF
```

Fig. 6. The core part of expandTestCase function

IV. EXPERIMENTAL RESULT

We design and implement the proposed algorithm. In order to evaluate the improvement, TFG is generated using the implemented system with various number of test cases. TABLE I shows the experimental result of executing time between the CPU and GPU. It shows that the GPU is faster than the CPU when the number of test case is larger than 70. Note that, the number of the node in the TFG determines the performance.

TABEL I
Executing Result

Number of Test case	Number of TFG node	CPU execution time (ms)	GPU execution time (ms)	Number of Blocks	Number of Threads
10	100	0.046195	0.176380	1	100
20	200	0.090523	0.191312	1	200
30	299	0.135318	0.239840	1	299
40	397	0.179180	0.240307	1	397
50	496	0.223042	0.203444	1	496
60	596	0.269703	0.292568	2	298
70	695	0.314032	0.314965	2	348
80	791	0.357427	0.315432	2	396
90	888	0.400822	0.324764	2	444
100	987	0.443284	0.343895	2	494
150	1478	0.665859	0.353694	3	493
200	1964	0.890301	0.330830	4	491
300	2930	1.330786	0.424153	6	489
400	3873	1.737674	0.482480	8	485
500	4806	2.348430	0.460540	10	481

Fig. 7 shows the graph according to the TABLE I. It shows the changing trends of CPU and GPU between test case and time. It shows that the line of GPU is almost a linear line that means as the test case number increases, the time GPU spent has a little growth. However, the CPU shows that the time CPU spent rises obviously as the test case number increases.

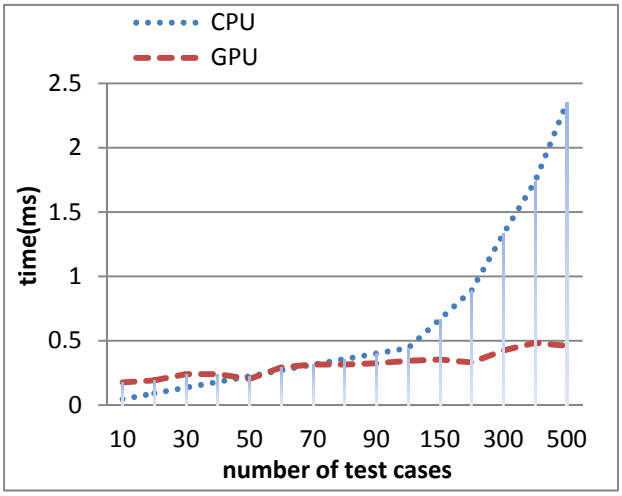

Fig. 7. Comparison between GPU and CPU

V. CONCLUSION

This paper proposed a parallel algorithm to generate GUI test cases using CUDA. Because of this kind of parallel way, the algorithm obtains efficient test result. It can generate the same amount of test cases with less time. The experimental result demonstrates that the proposed parallel algorithm that uses GPU is efficient than the algorithm that uses CPU when the number of test cases is larger than 70. We additionally tested the situation that the number of test cases is 100,000 and the number of nodes is almost 1,000,000. In this situation, GPU also shows better result and performs much more efficient than the CPU. As the future work, we will reduce memory used by TFG in order to process the larger number of test cases.

Acknowledgment

This paper was supported by Research Fund, Kumoh National Institute of Technology.

REFERENCES

[1] Kyung-Seok Park, Hyeon-Ju Yoon, Lee-Sub Lee, "GUI Test Case Paths Expansion Method using Reverse Engineering", Journal of Advanced Information Technology and Convergence, Vol.10, No.4, pp. 129-136, April 2012.

[2] Jae-Il Lee, Byoung-Woo Oh, "An Efficient Technique for Processing of Spatial Data Using GPU", The Journal of GIS Association of Korea, Vol.17, No.3, pp. 371-379, June 2009.

[3] Dae-Kwang Kim, Lee-Sub Lee, "A GUI Test Data Generation Method Using Reverse Engineering", Journal of Advanced Information Technology and Convergence, pp. 194-197, May 2011.

[4] Sungsoo Kim, Dongheon Kim, Sangkyu Woo, Insung Ihm, "Analysis of Programming Techniques for Creating Optimized CUDA Software", Journal of Computing Science and Engineering, Vol.16, No.7, pp. 775-787, July 2010.

[5] Kishore Kothapalli, Rishabh Mukherjee, M. Suhail Rehman, Suryakant Patidar, P. J. Narayanan, Kannan Srinathan, "A performance prediction model for the CUDA GPGPU platform", IEEE Conference Publications, pp. 16-19, Dec 2009.

[6] Jingweijia Tan, Nilanjan Goswami, Tao Li, Xin Fu "Analyzing Soft-Error Vulnerability on GPGPU Microarchitecture", IEEE Conference Publications, pp. 226-235, Nov 2011.

[7] In-Yong Jung, Chang-Sung Jeong "Parallel Connected-Component Labeling Algorithm for GPGPU Applications", IEEE Conference Publications, pp. 1149-1153, Oct 2010.

[8] Ryosuke Takahashi, Ushio Inoue "Parallel Text Matching Using GPGPU", IEEE Conference Publications, pp. 242-246, Aug 2012.

[9] Sharmistha, Madhur Amilkanthwar, Shankar Balachandran, "Augmentation of Programs with CUDA Streams", IEEE Conference Publications, pp. 855-586, July 2012.

[10] Alin Suciu, Lidia Zegreanu, Catalin Tudor Zima, "Statistical Testing of Random Number Sequences using CUDA", IEEE Conference Publications, pp. 26-28 , Aug 2010.

[11] Jiri Barnat, Petr Bauch, Lubos Brim, and Milan Ceska, "Employing Multiple CUDA Devices to Accelerate LTL Model Checking", IEEE Conference Publications, pp. 8-10, Dec 2010.