Scientific Research Publishing

Journal of Software Engineering and Applications, 2017, 10, 159-167
http://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

Detecting Bank Conflict of GPU Programs Using Symbolic Execution—Case Study

Koki Hamaya, Satoshi Yamane

Graduate School of Natural Science and Technology, Kanazawa University, Kanazawa, Japan
Email: khamaya@csl.ec.t.kanazawa-u.ac.jp, syamane@is.t.kanazawa-u.ac.jp

How to cite this paper: Hamaya, K. and Yamane, S. (2017) Detecting Bank Conflict of GPU Programs Using Symbolic Execution—Case Study. *Journal of Software Engineering and Applications*, **10**, 159-167.
https://doi.org/10.4236/jsea.2017.102009

Received: January 13, 2017
Accepted: February 18, 2017
Published: February 21, 2017

Abstract

GPU (Graphics Processing Unit) is used in various areas. Therefore, the demand for the verification of GPU programs is increasing. In this paper, we suggest the method to detect bank conflict by using symbolic execution. Bank conflict is one of the bugs happening in GPU and it leads the performance of programs lower. Bank conflict happens when some processing units in GPU access the same shared memory. Symbolic execution is the method to analysis programs with symbolic values. By using it, we can detect bank conflict on GPU programs which use many threads. We implement a prototype of the detector for bank conflict and evaluate it with some GPU programs. The result states that we can detect bank conflict on the programs with no loop regardless of the number of threads.

Keywords

Graphics Processing Units, GPU, Bank Conflict, Symbolic Execution, Model Checking

1. Introduction

Nowadays, GPGPU (General-purpose computing on GPU) is one of the most remarkable topics in the field of HPC and in various study fields [1]. Originally, GPU is used for graphics processing. Beside, GPGPU is to use GPU for other general purpose computation. The computation power of GPU achieves results greatly for not only graphics processing but also other objects.

If you want to make the most use of GPU, you should write GPU programs with considering the architecture of GPU. One of what you should do is not to make bank conflict. Bank conflict happens when some processing units in GPU access the same shared memory. When bank conflict happens, the program instruction executed in parallel is executed in sequential. As a result, it makes the performance lower. It is hard for programmers to find whether bank conflict

happens manually. If there is a bank conflict detector for GPU programs, debugging is easier.

In this paper, we propose the method to detect bank conflict. The method uses symbolic execution to analysis GPU programs. By executing symbolically, GPU programs using many threads can be executed with the number of states small and state explosion is lightened. We implement a prototype of our method and evaluate it with some tests.

2. Related Works

In Utah university, P. Li, G. Li and G. Gopalakrishnan study verification for GPU programs with symbolic execution [2]. In our study, we target GPU programs written in CUDA, the environment and language for GPU programs, and use ordinary symbolic execution. In Utah university's study, they target GPU programs written in LLVM, the intermediate representation compiled from CUDA codes, and use conclic execution. Conclic execution is the technique that programs are executed regarding a part of variables as symbolic values and other part of variables as concrete values. It can make the number of states smaller than symbolic execution.

In Imperial College London, A. Betts *et al.* study verification for GPU programs with Boogie [3]. Boogie is the verification language proposed by Microsoft Research [4]. They convert GPU programs to the intermediate representation written in Boogie language. Then Boogie verifier is used to check the satisfiability of the intermediate presentation.

In Twente University, M. Huisman *et al.* study verification for GPU programs with separation logic [5]. Strictly speaking, permission-based separation logic is used. They check this logic formula while permission for to read or to write is added when a memory is accessed and a barrier is executed.

There are two differences between these related works and our study. First, we target GPU programs written in CUDA instead of GPU programs written in LLVM. GPU programs written in CUDA is small than written in LLVM so that we can reduce the number of states. Second, we verify only bank conflict. Bank conflict happens in the same instruction, so we can assume all instruction is executed at the same time. By these two differences, the number of states in execution is smaller and the verification is easier.

3. GPU Software

3.1. GPU Hardware

GPU architecture is so unique compared to CPU [1]. GPU has hundreds or thousands of processing units. Because the number of processing units on GPU is huge, eight units make a set and GPU assigns this group to jobs. In this paper, based on names in NVIDIA, a GPU vendor, we call processing units on GPU SPs (Streaming Processors) and this group SM (Streaming Multiprocessor) (**Figure 1**).

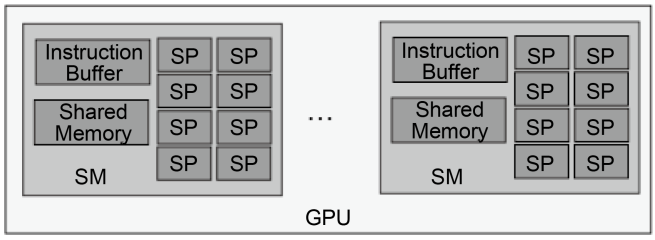

Figure 1. GPU Architecture.

One SM has eight SPs, one instruction buffer, and one shared memory. There is only one instruction buffer so that eight SPs in one SM execute the same instruction at the same time. Each instruction needs four clocks and Super-pipeline is employed in GPU. So each of eight SPs can execute four instructions at the same time. Therefore, one SM can execute 32 instructions at the same time and 32 threads make one warp. GPU handles many warps containing thousands or millions of threads and each warp is assigned to one SM.

3.2. GPU Programs

GPU Programs are written in the form of SIMD (Single-Instrument Multiple-Data) [1]. In this form, computers with multiple processing elements perform the same operation on multiple data points simultaneously. This form is suitable to GPU. There are some environments to make GPU programs in form of SIMD. One of the environments is CUDA. CUDA is the language and programming environment for GPU programs. we target GPU programs written in CUDA and execute it symbolically.

In CUDA, there are some built-in variables to write programs in form of SIMD. To make the description easier, we describe only one built-in variable, threadIdx. This variable indicates the index of threads. Enormous threads are made on GPU. So threads are managed in a three-dimensional such as x-axis, y-axis, and z-axis. ThreadIdx is defined as a structure containing the variables, x, y, and z. For example, when six threads are made and each size of dimensions is 3, 2, and 1, threadIdx's values are the followings. (x, y, z) = (1, 1, 1), (1, 2, 1), (2, 1, 1), (2, 2, 1), (3, 1, 1), (3, 2, 1) Each thread uses the same program but the processing is changed by using threadIdx. In detail, the condition of branch and the address of arrays contain threadIdx.

3.3. Shared Memory and Bank Conflict

Shared memory of SM has 32 banks (**Figure 2**). N address of shared memory belongs to (N% 32) bank as shown in **Figure 2**. For example, 0 address and 32 address belong to the same bank, 0 bank. The number of banks is 32 for the number of threads executed simultaneously is at most 32. One of important things is that at the same time, different bank can be accessed but the same bank cannot be accessed. If the same bank is accessed simultaneously, programs is executed not in parallel but sequentially.

Bank conflict is the bug happening by the property of bank. When multiple SP

tries to access the same bank of shared memory, completely parallel execution isn't achieved and the performance is lower. This phenomenon is called bank conflict. For example, bank conflict happens in Figure 3. The thread that threadIdx. x is 0 accesses 0 address and the thread that threadIdx is 16 accesses 32 address. So these two threads make bank conflict. This example is simple but as programs are more complicated, it is much harder to detect bank conflict manually. Therefore, a detector for bank conflict is needed.

4. Model Checking with Symbolic Execution

4.1. Satisfiability Modulo Theories

Before symbolic execution, we describe SAT and SMT. Given a proposal formula with proposal variables and logic operators, the problem to check whether a set of variables meeting this formula exists is SAT (Boolean or Propositional Satisfiability Testing). If the set exists, it is satisfiable. If not, it is unsatisfiable. A tool to check SAT is called SAT solver.

SMT is the extension of SAT with backgrounds in the area of mathematics. In SMT, check a first-logic formula with real number, bit operation, and data structure. If it is satisfiable, one example meeting this formula is got. Many SMT solvers have been developed and we use Z3.

Z3 is a SMT solver developed by Microsoft Research [6]. It has such theory solvers as linear arithmetic, bit-vectors, arrays, and tuples. So we decided to use it in our algorithm. There are two points to use Z3. First, we check whether a loop should be expanded in symbolic execution. Second, we check whether bank conflict happens on each symbolic state.

4.2. Symbolic Execution

Symbolic execution is one of methods to execute and analysis programs [7] [8] [9]. The main idea of symbolic execution is to use not actual data but symbolic

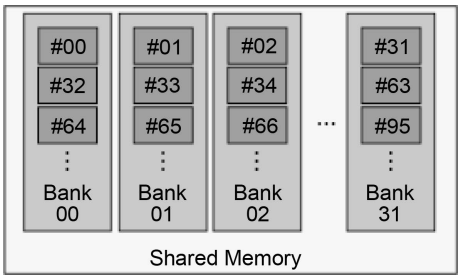

Figure 2. Shared memory and bank.

```
1 global void bankconflict ()
2 {
3       shared int smem[512];
4       int tid = threadIdx.x;
5       smem[tid * 2] = tid;
6 }
```

Figure 3. Example causing bank conflict.

values as inputs and to represent the values of program variables as symbolic formula. As a result, the final outputs calculated by programs represent functions with symbolic values. In symbolic execution, a state of programs in execute has the symbolic values of variables and the path condition. A path condition is a symbolic formula which means the condition to execute this state. Symbolic execution tree is made as a result of symbolic execution. It is an execution tree with symbolic formula.

4.3. Model Checking with Symbolic Execution

Model checking is one of formal methods to check whether a program meets a property. In model checking, use a model to represent the system and a property met in the system. Generally, a model is an automaton such as Kripke structure and a property is written in temporal logic.

In our study, a model is a symbolic execution tree and a property is written in temporal logic. For the states which access shared memory, the conjunction of the negation of the symbolic formula representing the property and the path condition is checked by SMT solver. If the conjunction is unsatisfiable, the property is met. If the conjunction is satisfiable, the property isn't met and we can get the counter-example. This counter-example is an example of an input which doesn't meet the property.

5. Our Algorithms

In our study, bank conflict on GPU programs is detected. To detect it, we use model checking with symbolic execution. There are some reasons to use it.

The first reason is to reduce the state explosion. Bank conflict happens between any pair of threads. The number of states is enormous considering all combinations of all threads. If we assume that the number of threads is n, the number of two-threads combinations is $n(n-1)/2$. In addition, a branch on programs makes more states and it is easy to guess that the state explosion gets worse. By using symbolic execution, only one pair of threads having threadIdx with symbolic values is needed. This is the main reason to use symbolic execution.

The second reason is that we target GPU programs written in CUDA. Inputs of programs is usually undecidable, so it is convenient to set symbolic values to inputs. If inputs aren't given, we can detect bank conflict.

5.1. Overview

The overview of our study is shown in **Figure 4**. First, we convert GPU programs to the control flow graph. Then, symbolic execution is done on the graph and the graph is converted to the symbolic execution tree. We will describe how to execute symbolically in the next subsection. Finally, for each state of the symbolic execution tree, we check whether bank conflict happens by using SMT solver. Strictly speaking, for each state of the tree, we give the conjunction of the path condition and the condition for detecting bank conflict to SMT solver. If

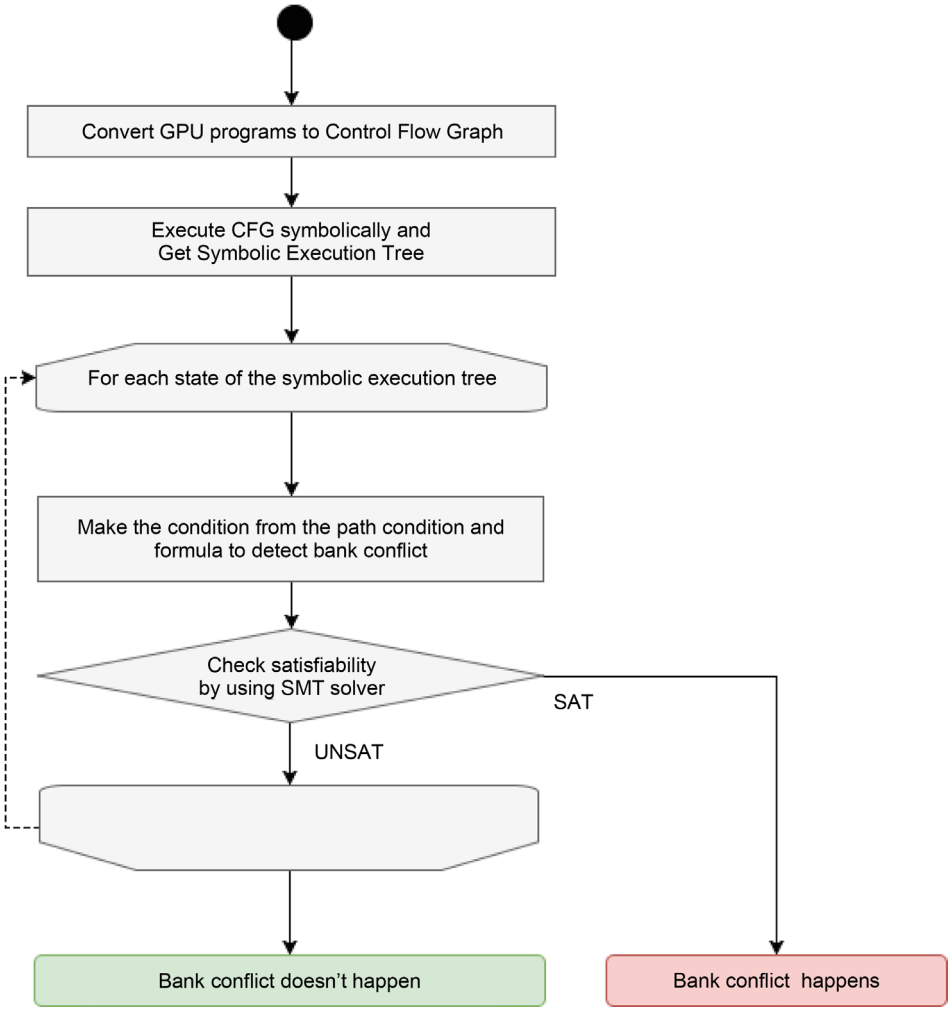

Figure 4. Overview of our algorithm.

the solver says satisfiable, bank conflict happens. If the solver says unsatisfiable, bank conflict doesn't happen. A condition for detecting bank conflict is shown in the later subsection.

5.2. Symbolic Executor

It describes symbolic executor in our study. It takes a control flow graph as inputs and outputs the symbolic execution tree. Each state of a symbolic execution tree contains a program counter, values of program variables, and a path condition. Initially, an initial state is made. This path condition is True and program variables are assigned different symbolic values. Symbolic execution is done in the following rules.

1) In an assignment expression, the path condition isn't changed but values of program variables are calculated.

2) In a branch expression if (e) S1 else S2, the state is divided into two state, "the state that the condition is true" and "the state that the condition is false". In "the state that the condition is true", the path condition pc is updated to pc $\wedge$ e. In "the state that the condition is false", the path condition pc is updated to pc

$\wedge$ e. In both states, the values of program variables aren't changed.

3) In a loop expression, the loop is expanded until the condition is met. SMT solver is used to check whether a loop is expanded. If the condition is satisfiable, expand the loop. If not, stop expanding the loop.

5.3. Model Checking

We check whether bank conflict happens in each state of the symbolic execution tree. For it, 1) we convert a path condition for one thread convert into the path condition for two threads and 2) we check satisfiability of the conjunction of two path conditions and the condition for detecting bank conflict.

1) Two path conditions are needed because bank conflict happens between two threads. The variables in CUDA are local variables, shared memory variables, device public variables, and built-in variables. Local variables and built-in variables are peculiar to one thread, so these variables of two threads has different symbolic values. Shared memory variables and device public variables are common in threads, so these variables have the same symbolic values.

2) The condition for detecting bank conflict is the conjunction of three conditions. The first condition is for two symbolic variables, *a_vaddr* and *b_vaddr*. These variables indicate the address of shared memory in which we check whether bank conflict happens. These variables are based on the state of symbolic execution tree. The second condition is the following.

$$0 \le tn \le \text{blockDim}/32,\ 0 \le a < 32,\ 0 \le b < 32 \tag{1}$$

$$\text{threadIdx}\_a = tn * 32 + a \tag{2}$$

$$\text{threadIdx}\_b = tn * 32 + b \tag{3}$$

The variable, threadIdx_*a*, indicates one of two threads and the variable, threadIdx_*b*, indicates the other. To consider threadIdx in one warp, two threadIdxs need to be chose in a warp. For it, introduce the variables, *tn*, *a*, and *b*. The third condition is the following.

$$\text{diff} = a\_vaddr - b\_vaddr,\ \text{diff}\,\%32 = 0 \tag{4}$$

This condition indicates that the two addresses of shared memory belong to the same bank. By this condition, we can check whether bank conflict happens. If SMT solver says satisfiable, bank conflict happens and the counter-example means the actual values of programs variables. If SMT solver says unsatisfiable, bank conflict doesn't happen.

6. Implementation

We develop a prototype of our model checker to detect bank conflict. We prepare seven GPU programs to evaluate the prototype. These programs are a part of test cases for GPU programs, Gklee Tests, published by Utah University [10]. We made experiments as the number of threads on GPU change to evaluate how a change of the number of threads affects the verification time. We consider two group of the result. The first group is the group that GPU programs contain no loop and other group is that contain a loop.

Figure 5 shows the result of evaluation for GPU programs containing no loop. Programs containing no loop are the five of seven GPU programs. Thanks to regard the number of threads on GPU as the symbolic value, as the number of threads increases, the number of states to be checked is fixed. Because the number of states is fixed, the verification time is also fixed. We achieve the aim that the state explosion is reduced if the number of threads increases.

Figure 6 shows the result of evaluation for GPU programs containing a loop. Programs containing a loop are the two of seven GPU programs. In this group, an increase in the number of threads makes the number of loop unrolling and leads to an increase in the number of states. It is found that we cannot solve the state explosion by loops. In our algorithm, a loop is expanded until the loop condition is unsatisfiable. If the loop condition is related to the number of threads, the number of loop unrolling is larger as the number of threads is larger. Hence we expect that these two programs had the loop involved in the number of threads and symbolic executor made a larger number of states than Group 1. Therefore, we need to handle a loop unrolling well to reduce the number of states.

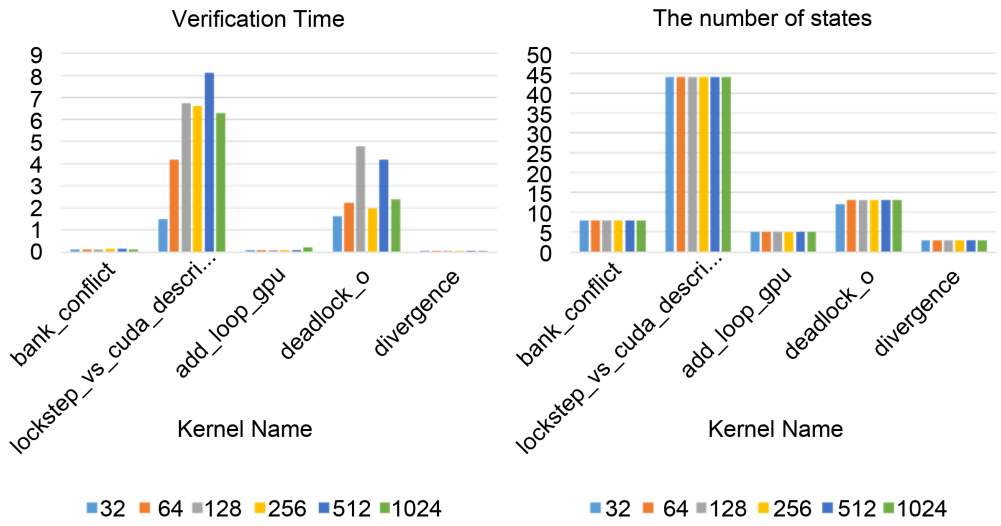

Figure 5. Group 1.

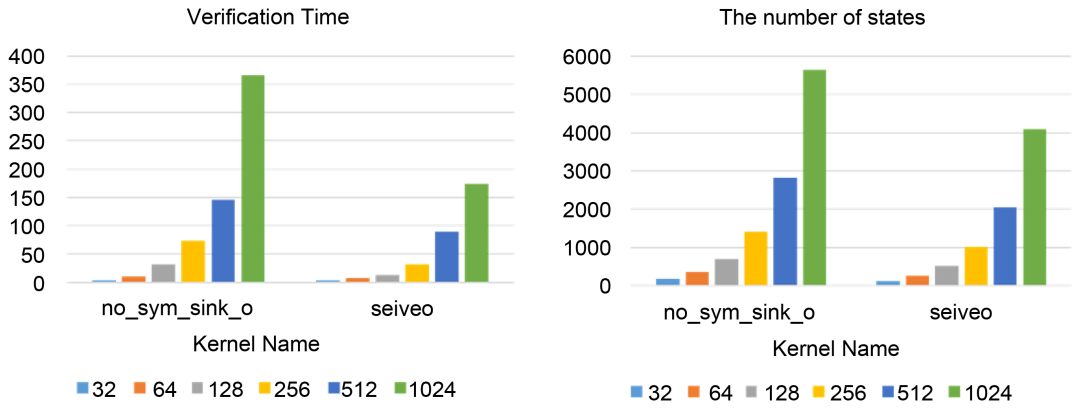

Figure 6. Group 2.

7. Conclusions

In the paper, we describe GPU architecture, model checking with symbolic execution, and, the method to detect bank conflict. In our method, we convert GPU programs written in CUDA to the control flow graph to execute symbolically and execute symbolically the programs on the control flow graph. Then we check whether bank conflict happens on each states of the symbolic execution tree given by a symbolic executor.

We evaluated the prototype of this method. The result shows that we can detect bank conflict on programs with no loop regardless of the number of threads. It means we achieve to avoid state explosion due to increasing of the number of threads. However, when we analyze the programs containing a loop, state explosion happens. Hence we need to reduce the number of states by handling a loop unrolling well.

As the future work, we try to think about the method to reduce the number of states by using counter-example guided abstraction refinement (CEGAR) [11]. CEGAR is one of good methods to check programs with handling a loop well. Besides, we try to detect other bugs on GPU programs.

References

[1] Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E. and Phillips, J.C. (2008) GPU Computing. *Proceedings of the IEEE*, **96**, 879-899. https://doi.org/10.1109/JPROC.2008.917757

[2] Li, P., Li, G. and Gopalakrishnan, G. (2014) Practical Symbolic Race Checking of GPU Programs. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, New Orleans, 16-21 November 2014, 179-190. https://doi.org/10.1109/SC.2014.20

[3] Betts, A., *et al.* (2015) The Design and Implementation of a Verification Technique for GPU Kernels. *ACM TOPLAS*, **37**, Article No. 10.

[4] Barnett, M., Evan Chang, B.-Y., DeLine, R., Jacobs, B. and Leino, K.R.M. (2005) Boogie: A Modular Reusable Verifier for Object-Oriented Programs. Vol. 4111, Springer, Berlin Heidelberg, 364-387.

[5] Huisman, M. and Matej, M. (2013) Specification and Verification of GPGPU Programs Using Permission-Based Separation Logic.

[6] De Moura, L. and Nikolaj, B. (2008) Z3: An Efficient SMT Solver. Vol. 4963, Springer, Berlin Heidelberg, 337-340. https://doi.org/10.1007/978-3-540-78800-3_24

[7] Sarfraz, K., Psreanu, C.S. and Visser, W. (2003) Generalized Symbolic Execution for Model Checking and Testing. Vol. 2619, Springer, Berlin Heidelberg, 553-568.

[8] Psreanu, C.S. and Willem, V. (2009) A Survey of New Trends in Symbolic Execution for Software Testing and Analysis. *International Journal on Software Tools for Technology Transfer*, **11**, 339-353. https://doi.org/10.1007/s10009-009-0118-1

[9] Cadar, C. and Koushik, S. (2013) Symbolic Execution for Software Testing: Three Decades Later. *Communications of the ACM*, **56**, 82-90. https://doi.org/10.1145/2408776.2408795

[10] Geof23/GkleeTests. https://github.com/Geof23/GkleeTests

[11] Beyer, D., *et al.* (2007) The Software Model Checker BLAST. *International Journal on Software Tools for Technology Transfer*, **9**, 505-525.