Exploiting GPU Architectures for Dynamic Invariant Mining

Nicola Bombieri*, Federico Busato*, Alessandro Danese*, Luca Piccolboni* and Graziano Pravadelli*†
*Department of Computer Science, University of Verona, Italy. Email: name.surname@univr.it
† EDALab s.r.l., Italy. Email: name.surname@edalab.it

***Abstract*—Dynamic mining of invariants is a class of approaches to extract logic formulas from the execution traces of a system under verification (SUV), with the purpose of expressing stable conditions in the behaviour of the SUV. The mined formulas represent likely invariants for the SUV, which certainly hold on the considered traces, but there is no guarantee that they are true in general. A large set of representative execution traces must be analysed to increase the probability that mined invariants are generally true. However, this becomes extremely time-consuming for current sequential approaches when long execution traces and large set of SUV variables are considered. To overcome this limitation, the paper presents a parallel approach for invariant mining that exploits GPU architectures for processing an execution trace composed of millions of clock cycles in few seconds.**

I. Introduction

Invariant mining is a technique to extract logic formulas that hold between a couple (or several couples) of points in an implementation. Such formulas express stable conditions in the behaviour of the system under verification (SUV) for all its executions, which can be used to analyse several aspects in verification of SW programs and HW designs, at different abstraction levels. For example, invariant mining has been applied for analysis of dynamic memory consumption [1], static checking [2], detection of race conditions [3], identification of memory access violations [4], test generation [5], mining of temporal assertions [6] and bug catching in general [7].

Both static and dynamic approaches exist for mining invariants. The first exhaustively and formally explore the state space of the SUV [8], [9], but they work well for relatively small/medium size implementations. Moreover, they require the source code of the SUV is available. When larger designs are considered, dynamic techniques represent a not exhaustive but more scalable solution, since they rely on simulation rather than formal methods [7], [10], [11], [12]. Moreover, these approaches are the unique alternative when the source code of the SUV is non available. In fact, they generally work by analysing a set of execution traces of the SUV and searching for counterexamples of the logic formulas that represent the desired invariant candidates. However, at the end of the analysis, survived candidates are *likely invariants*, i.e., formulas that are only statistically true on the SUV, because they have been proved to hold only on the analysed traces. For this reason, to increase the degree of confidence on likely invariants, a large and representative set of execution traces must be analysed by dynamic approaches. Unfortunately, for complex HW designs this could require to elaborate thousands of execution traces, including millions of clock cycles, and predicating over hundreds of variables, which becomes an unmanageable time-consuming activity for existing approaches.

The solution we propose to speed-up the mining process is to move from a sequential to a parallel implementation of likely invariant miners, such that general-purpose computing on graphics processing units (GPGPU) can be exploited to significantly reduce the time required for processing a large number of execution traces composed of millions of clock cycles. A first parallel approach for invariant mining has been presented in [13] showing sensible improvements with respect to Daikon [10], one of the most popular sequential miners. In this paper, we propose an alternative parallel algorithm that greatly benefits from advanced graphics processing unit (GPU) programming techniques, such that the memory throughput of the GPU is significantly improved. In this way, as reported in the experimental results, the overall performance of the mining algorithm are increased up to three orders of magnitude with respect to [13].

The rest of the paper is organized as follows. Section II briefly summarizes the main concepts on the GPU programming model and defines some preliminary concepts. Section III describes the proposed parallel approach for dynamic invariant mining. Finally, Section IV and Section V are devoted, respectively, to experimental results and concluding remarks.

II. Background

A. CUDA programming model for GPUs

Compute Unified Device Architecture (CUDA) is a programming model developed by NVIDIA to provide a programming interface to GPU devices [14]. Through API function calls, called *kernels*, and language extensions, CUDA allows enabling and controlling the offload of compute-intensive routines. A CUDA kernel is executed by a *grid* of *thread blocks*. A thread block is a batch of threads that can cooperate and synchronize each other via shared memory and barriers. GPU architectures provide high memory bandwidth at the cost of a high access latency. GPUs achieve full memory bandwidth and hide memory latency through the concept of memory coalescing that refers to combine multiple continuous memory accesses into a single transaction. Achieving memory coalescing is one of the main strategic techniques in GPU programming to sensibly improve the performance of a parallel application.

B. Preliminary definitions

The following definitions concerning *execution traces* and *likely invariants* are necessary to describe how the mining

978-1-4673-7166-7/15/$31.00 ©2015 IEEE

approach presented in Section III works.

Definition 1. Given a finite sequence of simulation instants $\langle t_1, ...t_n \rangle$ and a set of variables $\mathcal{V}$ of a model $\mathcal{M}$, an *execution trace* of $\mathcal{M}$ is a finite sequence of pairs $\tau = \langle (\mathcal{V}_1, t_1), ...(\mathcal{V}_n, t_n) \rangle$, where $\mathcal{V}_i = eval(\mathcal{V}, t_i)$ is the value of variables in $\mathcal{V}$ at simulation instant t_i.

Definition 2. Given a model $\mathcal{M}$ and the corresponding sets of variables $\mathcal{V}$ and execution traces $\mathcal{T}$, a *likely invariant* for $\mathcal{M}$ is a logic formula over $\mathcal{V}$ that holds throughout each $\tau \in \mathcal{T}$.

III. Invariant mining

The main mining function, in its sequential form, is reported in Algorithm 1. The inputs of the function are represented by an execution trace τ of the SUV, an invariant template set $\mathcal{I}$, and a variable dictionary $\mathcal{D}$. The dictionary contains tuples of different arity composed by all the possible combinations of the variables $\mathcal{V}$ of the SUV. Such tuples represent the actual parameters to be substituted inside the formal parameters of the invariant templates during the mining phase.

The algorithm extracts all likely invariants for τ that correspond to logic formulas included in $\mathcal{I}$, by substituting in the elements of $\mathcal{I}$ all the possible tuples of $\mathcal{V}$ belonging to $\mathcal{D}$, according to the respective arity. More precisely, the *check_invariant* function (line 5) checks if a specific template INV, instantiated with the current tuple of variables TUPLE, holds at simulation time INSTANT. When a counterexample is found for INV, it is removed from the template set (line 6) for the current tuple of variables. If all elements of the template set are falsified (line 8), the algorithm restarts by considering the next tuple in the dictionary, by skipping the remaining simulation instants of τ. At the end, the algorithm collects all the pairs composed by the the survived templates and the corresponding tuples of the variable dictionary (line 11). The instantiation of the tuples in the survived templates represent the final set of likely invariants for τ. The current implementation supports the invariant template sets reported in Table I. Boolean and numeric templates include, respectively, only Boolean variables and numeric variables.

Algorithm 1 The invariant mining algorithm.

```
    SEQUENTIAL_MINING($\mathcal{D}$, $\mathcal{I}$, $\tau$) return result
 1: for all TUPLE $\in \mathcal{D}$ do
 2:     template_set = $\mathcal{I}$
 3:     for all INSTANT $\in \tau$ do
 4:         for all INV $\in$ template_set do
 5:             if $\neg$check_invariant(INV,TUPLE,INSTANT) then
 6:                 template_set = template_set $\setminus$ INV
 7:         end
 8:         if template_set = $\emptyset$ then
 9:             break
10:     end
11:     result = result $\cup$ $\langle$TUPLE, template_set$\rangle$
12: end
```

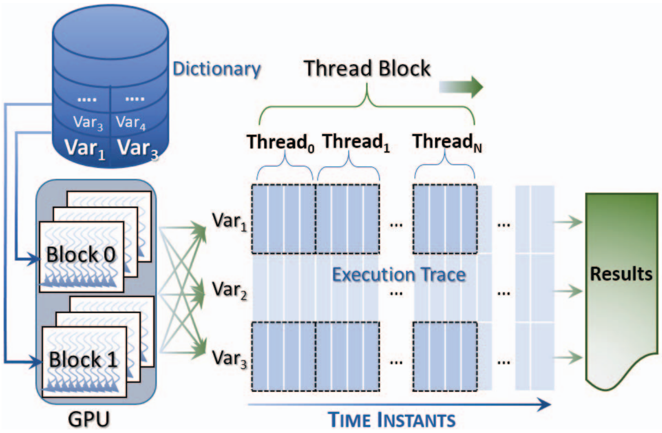

FIG. 1: *Overview of block mapping and vectorized accesses for the parallel algorithm on GPU.*

The proposed algorithm has a worst-case time complexity equal to $\mathcal{O}(|\mathcal{V}|^K \cdot |\tau| \cdot |\mathcal{I}|)$, where $\mathcal{V}$ is the number of considered variables, K is the arity of the invariant template belonging to $\mathcal{I}$ with the highest arity, $|\tau|$ is the number of simulation instants in the execution trace τ, and $|\mathcal{I}|$ is the number of invariant templates included in $\mathcal{I}$.

A. The parallel implementation for GPUs

The mining approach reported in Algorithm 1 is well suited for parallel computation. In fact, the problem can be easily decomposed in many independent tasks, each one having regular structure and fairly balanced workload. For this reason we implemented a parallel version of the mining algorithm, called *Mangrove*. It implements the mining algorithm with the aim of exploiting the massive parallelism of GPUs and, at the same time, an inference strategy to reduce redundant checking of invariants, as explained in Section III-B. In addition, since reading the execution trace from the mass storage and moving it to the GPU device for the mining phase is computationally time consuming, *Mangrove* implements a strategy to overlap data transfers and mining phase, as explained in Section III-C.

Fig. 1 shows an overview of the GPU kernel organization and, in particular, how thread blocks are mapped to the trace variables for the reading and mining phases. Consider a matrix representing the execution trace, in which the columns represent the execution time instants and the rows hold the variable values (for the sake of clarity, the execution trace of the example in Fig. 1 consists of three variables). Each thread block is mapped to (i.e., it performs the reading and mining phase on) one, two, or three matrix rows if the mining process performs over a unary, binary, or ternary template, respectively (i.e., each thread block is mapped to a different entry of the variable dictionary). In each block, threads communicate and synchronize through shared memory. As for the standard characteristics of the GPU architectures, such hardware-implemented operations are extremely fast and their overhead is negligible. Communication and synchronization among block threads allow avoiding redundant checking of already falsified invariants and stopping the computation of the whole block as soon as all invariants for a particular set of variables have been falsified.

	BOOLEAN			NUMERIC		
	UNARY	BINARY	TERNARY	UNARY	BINARY	TERNARY
TEMPLATE SET I	true, false				$=, \neq, <, >, \leq, \geq$	
TEMPLATE SET II	true, false	$=, \neq$	$\mathrm{Var}_1 = \mathrm{Var}_2 \,\mathrm{AND}\, \mathrm{Var}_3$ $\mathrm{Var}_1 = \mathrm{Var}_2 \,\mathrm{OR}\, \mathrm{Var}_3$ $\mathrm{Var}_1 = \mathrm{Var}_2 \,\mathrm{XOR}\, \mathrm{Var}_3$	$\mathrm{Var} = \mathrm{Const}$ $\mathrm{Var} \neq \mathrm{Const}$ $\mathrm{Var} < \mathrm{Const}$ $\mathrm{Var} \leq \mathrm{Const}$	$\mathrm{Var}_1 = \mathrm{Var}_2$ $\mathrm{Var}_1 \leq \mathrm{Var}_2$ $\mathrm{Var}_1 < \sqrt{\mathrm{Var}_2}$ $\mathrm{Var}_1 = \log \mathrm{Var}_2$ $\mathrm{Var}_1 < \mathrm{Var}_2 + 1$ $\mathrm{Var}_1 = \mathrm{Var}_2 * 2$	$\mathrm{Var}_1 = {\mathrm{Var}_2}^{\mathrm{Var}_3}$ $\mathrm{Var}_1 = \min(\mathrm{Var}_2, \mathrm{Var}_3)$ $\mathrm{Var}_1 = \max(\mathrm{Var}_2, \mathrm{Var}_3)$ $\mathrm{Var}_1 < \mathrm{Var}_2 * \mathrm{Var}_3$ $\mathrm{Var}_1 \leq \mathrm{Var}_2 + \mathrm{Var}_3$

TABLE I: *Template sets considered by the miner.*

For each block, each thread is mapped to four columns. This allows enabling the *vectorized* accesses [15] of threads to the variable values in memory. In particular, each thread loads four consecutive 32-bit words instead of a single word to improve the memory bandwidth between DRAM and thread registers. In addition, the Boolean and numeric variables included in the variable dictionary are organized over bit and float arrays in *row-major order*. This allows the full coalescing of memory accesses by the GPU threads in the mining phase.

Mangrove computes the mining process by elaborating, in sequence, the unary templates, the binary templates, and, finally, the ternary templates reported in Table I.

B. Optimization of the variable dictionary

The variable dictionary consists of a data structure that initially stores, for each invariant template, all the tuples of variables that must be substituted as actual parameters in the template during the mining phase. However, at run time, *Mangrove* implements some optimizations in the variable dictionary, to increase the efficiency of the mining. In particular, *Mangrove* optimizes the variable dictionary by discarding a tuple for a template when the answer of the relative checking phase can be derived from the results obtained from previous iterations of the mining procedure. This allows saving time by avoiding redundant elaborations, as explained in the next paragraphs:

- The result of the mining over unary templates is exploited during the mining of binary templates. As a simple example, *Mangrove* searches for any Boolean variable, var_a, whose value is always equal (or always different) to any other Boolean variable, var_b. If such a condition occurs, the generation of the entry $< var_a, var_b >$ in the dictionary can be avoided since it is redundant.
- The result of the mining over unary and binary templates is used during the mining of ternary templates. For example, by considering the ternary mining phase on Boolean variables, the goal is to figure out which operator $op \in \{\mathrm{AND}, \mathrm{OR}, \mathrm{XOR}\}$ can be validated over three different variables (e.g., var_a, var_b, and var_c). Through the already extracted unary and binary invariants, *Mangrove* automatically infers some ternary invariants without applying the checking procedure throughout the execution traces. For instance, the ternary invariant $(var_a = var_b$ $\mathrm{AND}\ var_c)$ reduces to check whether the binary invariant $(var_a = var_b)$ occurs when $(var_b = var_c)$ holds. Similarly $(var_a = var_b \ \mathrm{XOR}\ var_c)$ reduces to check $(var_a \neq var_c)$ when var_b is constantly set to *true*.

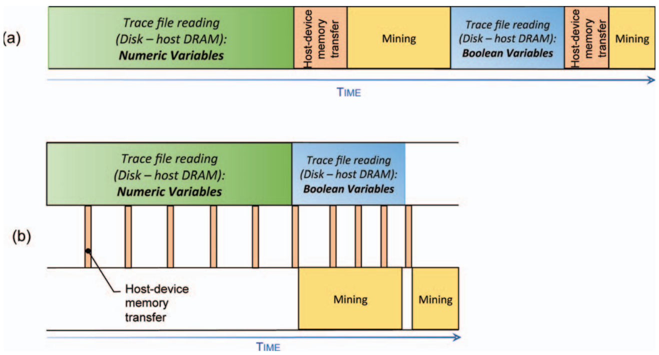

FIG. 2: *The invariant mining phases (a), and the overlapped implementation of* Mangrove *for GPUs (b).*

C. Data transfer and overlapping of the mining phase

The invariant mining process on the GPU consists of three main phases showed in Fig. 2(a): (i) reading of the execution trace from the mass storage (disk) and data storing in the host DRAM memory; (ii) data transfer from the host to the memory of the GPU; (iii) elaboration in the GPU device. The three steps work first on the numeric variables and then they are repeated for the Boolean variables. The time spent for such three phases and, in particular, the percentage of time spent by each phase over the total execution time depends on the invariant template and on the hardware characteristics. For instance, considering a magnetic mass storage disk, the reading and data transfer phases spend around the 80% and 20% over the total time, respectively while the mining time is negligible for unary templates. The three phases spend around 70%, 20%, and 10%, respectively, over the total time for binary templates. The percentage of the reading phase sensibly decreases in case of solid-state disks (SSDs).

Mangrove implements the invariant mining by overlapping the three phases as shown in Fig. 2(b). This allows totally hiding the cost of host-device data transfers and partially hiding the cost of the mining elaboration. Moreover, *Mangrove* implements the data transfer overlapping through asynchronous kernel invocations and memory copies (i.e., `cudaMemcpyAsync` in CUDA). Finally, a specific optimization has been implemented for Boolean variables: *Mangrove* stores the values of Boolean variables in arrays of bits to reduce the memory occupation (e.g., 5,000,000 values of a Boolean variable are stored in 600 KB). In addition, this array-based representation allows using bitwise operations to concurrently elaborate 32 Boolean values in a single chunk, thus speeding up the mining phase.

	Length	Boolean Vars	Numeric Vars	Invariants (Temp. set I)	Invariants (Temp. set II)
TRACE 1	5,000,000	15	15	0	0
TRACE 2	5,000,000	15	15	142	964
TRACE 3	5,000,000	50	50	0	0
TRACE 4	5,000,000	50	50	1,788	42,371

Table II: *Characteristics of execution traces.*

		Daikon[10]	Sequential[13]	Parallel[13]	Mangrove
Template Set I	TRACE 1	103 s	< 1 ms	116 ms	< 1 ms
	TRACE 2	170 s	4,629 ms	116 ms	17 ms
	TRACE 3	287 s	2 ms	369 ms	< 1 ms
	TRACE 4	1366 s	52,160 ms	457 ms	182 ms
Template Set II	TRACE 1	2 m 34 s	22 ms	352 ms	< 1 ms
	TRACE 2	5 m 47 s	11 m 0 s	1,751 ms	140 ms
	TRACE 3	8 m 23 s	119 ms	3,145 ms	< 1 ms
	TRACE 4	32 m 54 s	7 h 45 m	71,314 ms	4,577 ms

Table III: *Comparison of the execution times with respect to state-of-the-art approaches.*

IV. Experimental results

Experimental results have been run on a NVIDIA Kepler GeForce GTX 780 device with 5 GHz PCI Express 2.0 x16, CUDA Toolkit 7.0, AMD Phenom II X6 1055T 3GHz host processor, and the Debian 7 Operating System. To evaluate the efficiency of *Mangrove* experiments have been conducted on different kinds of execution traces, whose characteristics are summarized in Table II. Traces have been synthetically generated such that they expose from no invariant to thousands of likely invariants by considering the template sets reported in Table I. They also differ in terms of number of considered SUV variables. These are the two parameters that most influence, together with the length of the trace, the execution time of the mining algorithm. Indeed, higher is the number of likely invariants exposed by the execution traces, higher is the time spent for their extraction.

The efficiency of *Mangrove* has been compared against the sequential mining approaches implemented, respectively, in [10] and in [13], and the parallel implementation proposed in [13]. Table III shows the execution time required to extract the likely invariants according the first and second template sets on the traces reported in Table I. For the parallel approaches, the times include the overhead introduced for data transfer between host and device. *Mangrove* provides the best results in all datasets by executing up to four orders of magnitude faster than the sequential state-of-the-art tool Daikon[1]. Compared to the more recent approach for GPUs described in [13], *Mangrove* executes up to three orders of magnitude faster[2]. The improvements achieved in *Mangrove* with respect to the parallel approach implemented in [13] are due to the implementation of a more efficient strategy for mapping thread blocks to entries of the variable dictionary, and to the vectorized accesses that best exploit the memory coalescence and the high memory throughput. These aspects are critical to improve the performance, since the memory bandwidth may limit the concurrent memory accesses. Table III shows that *Mangrove* is efficient also when no invariant can be mined (Traces 1 and 3) thanks to the capability of early terminating the search on a trace as soon as all templates have been falsified. On the contrary, the parallel implementation proposed in [13] always requires to analyse the whole trace to identify the absence of likely invariants, thus wasting time.

V. Concluding remarks

The paper presented *Mangrove*, a parallel approach for mining likely invariants by exploiting GPU architectures. Advanced GPU-oriented optimizations and inference techniques have been implemented in *Mangrove* such that execution traces composed of millions of clock cycles can be generally analysed in less than one second searching for thousands of likely invariants. Experimental results have been conducted on execution traces with different characteristics, and the proposed approach has been compared with sequential and parallel implementations of the most promising state-of-the-art invariant miners. Analysis of the results showed that Mangrove outperforms existing tools.

Acknowledgment

This work has been partially supported by the EU project CONTREX (FP7-2013-ICT-10-611146).

References

[1] V. Braberman, D. Garbervetsky, and S. Yovine, “A static analysis for synthesizing parametric specifications of dynamic memory consumption,” *J. of Object Technology*, vol. 5, no. 5, pp. 31–58, 2006.

[2] J. W. Nimmer and M. D. Ernst, “Invariant inference for static checking: An empirical evaluation,” in *Proc. of ACM FSE*, 2002, pp. 11–20.

[3] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, “Eraser: A dynamic data race detector for multithreaded programs,” in *ACM Trans. on Computer Systems*, 1997, pp. 391–411.

[4] R. Hastings and B. Joyce, “Joyce. purify: Fast detection of memory leaks and access errors.” in *Proc. of USENIX*, 1991.

[5] C. Csallner and Y. Smaragdakis, “Check ’n’ crash: Combining static checking and testing,” in *Proc. of ACM/IEEE ICSE*, 2005, pp. 422–431.

[6] A. Danese, T. Ghasempouri, and G. Pravadelli, “Automatic extraction of assertions from execution traces of behavioural models,” in *Proc. of ACM/IEEE DATE*, 2015, pp. 1–6.

[7] S. Hangal and M. S. Lam, “Tracking down software bugs using automatic anomaly detection,” in *Proc. of ACM/IEEE ICSE*, 2002, pp. 291–301.

[8] C. Flanagan, R. Joshi, and K. R. M. Leino, “Annotation inference for modular checkers,” *Inf. Process. Lett.*, vol. 77, no. 2-4, pp. 97–108, 2001.

[9] N. Tillmann, F. Chen, and W. Schulte, “Discovering likely method specifications,” in *Formal Methods and Software Engineering*, ser. LNCS, Z. Liu and J. He, Eds. Springer, 2006, vol. 4260, pp. 717–736.

[10] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, “The Daikon system for dynamic detection of likely invariants,” *Science of Computer Programming*, vol. 69, no. 1, pp. 35–45, 2007.

[11] S. Hangal, S. Narayanan, N. Chandra, and S. Chakravorty, “IODINE: a tool to automatically infer dynamic invariants for hardware designs,” in *Proc. of ACM/IEEE DAC*, 2005, pp. 775–778.

[12] R. D. Marat Boshernitsan and A. Savoia, “From daikon to agitator: lessons and challenges in building a commercial tool for developer testing,” in *Proc. of ISSTA*, 2006, pp. 169–180.

[13] A. Danese, L. Piccolboni, and G. Pravadelli, “A parallelizable approach for mining likely invariants,” in *Proc. of ACM/IEEE CODES+ISSS*, 2015.

[14] “http://docs.nvidia.com/cuda.”

[15] J. Luitjens, “CUDA pro tip: Increase performance with vectorized memory access,” http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-increase-performance-with-vectorized-memory-access/, Dec. 2013.

[1] For a fair comparison, Daikon has been configured to search only for the invariants specified in the first and second template sets.

[2] The approach in [13] has been extended in order to support also the template set II.