Journal of Systems Architecture 127 (2022) 102518

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Symbolic identification of shared memory based bank conflicts for GPUs

Adrian Horga [a,*], Ahmed Rezine [a], Sudipta Chattopadhyay [b], Petru Eles [a], Zebo Peng [a]

[a] Linköping University, Sweden
[b] Singapore University of Tech. and Design, Singapore

ARTICLE INFO

Keywords:
GPU
Shared memory
Formal verification
Performance evaluation
Software security

ABSTRACT

Graphic processing units (GPUs) are routinely used for general purpose computations to improve performance. To achieve the sought performance gains, care must be invested in fine tuning the way GPU programs interact with the underlying architecture, accounting for the shared memory bank conflicts and the entailed shared memory transactions. Uncovering inputs leading to particular bank conflicts can turn out to be quite hard given the intricacy of the access patterns and their dependence on the inputs. We propose a symbolic execution based framework to systematically uncover shared memory bank conflicts, to propose inputs to realize a given number of shared memory transactions, and to refute the existence of such inputs if the number of shared memory transactions is impossible to achieve during the execution. This allows programmers to more formally reason about the shared memory conflicts and to validate their impact on performance and security. We have implemented our approach and report on our experiments to explore its usefulness towards performance enhancement and quantifying shared memory side-channel leakage in security applications.

1. Introduction

Graphic processing units (GPUs) are routinely used for parallel general purpose applications as they can leverage on the possibility to run programs on a large number of cores sharing several levels of memory. Cores in modern GPUs are less complex and slower than those found in modern CPUs. However, their sheer number and the possibility they have to efficiently switch among threads and to share data allows them to trade latency for throughput and to outperform modern CPUs on suitable applications.

Programming these platforms requires to fine tune the way data is partitioned, transferred and accessed by the large number of identical program instances. In particular, the way the GPU's shared memory is accessed can have an important impact on performance. Shared memory is partitioned into regions (so called banks) that can be separately and simultaneously accessed. Simultaneous accesses (so called bank conflicts) to the same bank have to be sequentialized into shared memory transactions yielding an important performance penalty. Works in [1–3] show that shared memory bank conflicts can have a direct impact on program performance through increased latency. The results in [2] show that, after a certain threshold, the shared memory bank conflicts require a latency beyond even the latency of reading directly from the GPU's global memory.

Traditionally, GPUs were used for graphical applications where shared memory was accessed according to fixed patterns that did not depend on the inputs. For such applications, it was enough to run the program on some arbitrary valid input to deduce the patterns and to fine tune the shared memory access to avoid bank conflicts. This is not anymore the case in modern applications of GPU platforms as they are being adopted for general purpose applications ranging from financial optimization [4] to security [5,6], and from graph manipulation [7,8] to machine learning [9–11]. Running GPU programs on concrete input values, which can consist of large arrays, to profile the way shared memory is accessed is not viable. For such programs, several shared memory access patterns may turn out to be very hard to uncover and their significant impact on performance will likely remain unnoticed during testing.

We propose a framework to systematically uncover inputs resulting in given shared memory access patterns. This framework allows us to ensure the absence of, or to produce, concrete inputs that lead to a given number of shared memory transactions resulting from the underlying bank conflicts. For this, we propose to augment symbolic execution for GPU programs to reason about the possibilities of shared memory bank conflicts and the patterns in which shared memory is accessed. The idea is to encode conflicts between simultaneous accesses to shared memory and to leverage on existing Satisfiability Modulo Theories (SMT) solvers to establish impossibility of the conflicts or to propose concrete inputs that will lead to such conflicts. We propose

* Corresponding author.
E-mail addresses: adrian.horga@liu.se (A. Horga), ahmed.rezine@liu.se (A. Rezine), sudipta_chattopadhyay@sutd.edu.sg (S. Chattopadhyay), petru.eles@liu.se (P. Eles), zebo.peng@liu.se (Z. Peng).

https://doi.org/10.1016/j.sysarc.2022.102518
Received 15 October 2021; Received in revised form 14 April 2022; Accepted 15 April 2022
Available online 20 April 2022

two feasible encodings: (1) a direct encoding that completely delegates the check to the underlying SMT solver and (2) an on-demand and incremental encoding that allows us to direct the search for feasibility and to cache intermediary results. We have implemented the approach on top of the CUDA [12] symbolic execution tool GKLEE [13]. We have tested our approach in the context of two important applications: performance evaluation and security analysis. For performance evaluation, we were able to deduce inputs that exhibit the best or worst numbers of shared memory transactions. We have also tested these inputs and validated the results on a real platform. For security analysis, we could leverage our approach on the possibility to refute the existence of inputs resulting in given shared memory conflicts. This allows us to establish bounds on the amount of information leaked by bank conflicts in GPU implementations of cryptographic algorithms.

To summarize, the contributions of this work are:

- We augment the symbolic execution for GPU programs with the possibility to establish, or refute, the possibility of shared memory bank conflicts.
- We propose and test three approaches to symbolically encode feasibility of numbers of shared memory transactions.
- We report on our experiments to systematically uncover possible bank conflicts for GPU programs and on our validation on a real GPU platform.
- We report on our experiments to establish bounds on the amount of information leaked by bank conflicts for two GPU-implementations of the AES cryptographic algorithm.

Shared memory is a central component of most modern GPU platforms. Leveraging on it to speed up memory accesses opens the possibility of bank conflicts and varying numbers of entailed shared memory transactions. Our contributions are valid for any modern GPU with a shared memory.

Our implementation and all experimental data are publicly available here: https://bitbucket.org/AdrianHorga/gpu_symbolicbankconflicts.

Related work. Shared memory bank conflicts have an important impact on latency and performance of GPU programs [1–3]. Typically, programmers use profilers [14,15] to track bank conflicts during concrete executions. Importantly, this only explores the behavior corresponding to a single concrete input and, therefore, only a tiny portion of the behavior of programs where shared memory accesses do depend on the input. This is not acceptable for a large class of applications where performance and/or security matters, including real-time [16,17], control [18] or cryptographic applications [5,6]. With the widespread usage of GPU platforms this limitation becomes more acute.

Work has been done on estimating program performance on GPU platforms [19–23]. The works in [19,20] approximately model the performance of GPU architecture. Other approaches on estimating GPU performance include machine-learning-based methods [21,22] and measurement-based methods [23]. In contrast to these works, our work has a significant testing flavor. For example, unlike the works focusing on empirical performance modeling, we can generate a concrete input that leads to a number of shared memory transactions induced by the underlying bank conflicts. We can also guarantee, through the leverage on symbolic execution and SMT solvers, that a worst case performance path is possible.

Symbolic execution explores one control path at a time. Following such a path, pairs of memory accesses in the same warp and at the same instruction can be compared to identify the possibility of a bank conflict. The exploration is however directed by the control path, not by the bank conflicts and the resulting shared memory transactions. For instance, the work in [13] only reports on the possibility of a bank conflict between two synchronization barriers. It does not explore possible simultaneous bank conflicts and resulting memory transactions. Simply answering whether there are bank conflicts at all between two barriers is not enough to explore or refute the possibility of bank conflicts leading to given numbers of shared memory transactions. The work in [24] also checks the possibility of a bank conflict due to shared memory accesses performed by an arbitrary pair of threads. For this, a path condition for one thread is duplicated to represent the path condition of a second thread in the same warp. This can be challenging due to branch divergence. In addition, the work only attempts to answer whether some bank conflict is present. It does not deduce the resulting number of shared memory transactions. Furthermore, the results (presence of bank conflicts) are not validated on a real GPU platform. All other applications of formal methods to GPU programs that we are aware of target checking race conditions, branch divergence or assertion violations. For instance the work in [25] analyzes arbitrary pairs using an over-approximation that havocs the shared state while the work in [26] adopts an under-approximation of the number of context switches. The work in [27] adapts permission based reasoning and requires annotation. The annotations are then validated to establish freedom from race conditions or assertion violation. Recently, the LLVM based static analysis proposed in [28] targets the problem of identifying uncoalesced memory accesses. This is a related, but different, problem from identifying shared memory bank conflicts and the resulting number of shared memory transactions. Our approach leverages on symbolic execution but focuses on tracking and checking the possibility of bank conflicts and the resulting shared memory transactions. In addition, if the targeted number of memory transactions is possible, our approach can supply a witness.

In [29], shared memory is proven to be a possible side channel for GPU cryptographic implementations. The work has since been expanded [30] to test multiple NVIDIA GPU generations. We aim to quantify the security impact of such a side channel through our approach. Work exists on quantifying side-channel leakage via cache behaviors for CPU implementations [31–34]. Shared memory in a GPU is a user controlled memory, as opposed to the cache. The specific behavior of shared memory compared to cache behavior requires a different modeling approach. Furthermore, the existence of multiple threads running in lockstep (in a warp) and in parallel during GPU program execution introduces further complexity in creating such a model when compared to cache models.

Our previous work [35] has targeted such a side channel through the use of genetic-algorithm-powered test input generation. Our current approach to expose the side channel differs from our previous work in several ways. First, by using a white box testing approach as opposed to a black box testing approach. Second, by leveraging the power of a symbolic execution tool and SMT solvers as opposed to genetic algorithms for generating test inputs. The third and most important difference in approaches is that the current approach can provide a more precise input selection and pruning method. This difference is important since it can help to *prove* whether certain bytes of a key can/cannot ever produce a targeted number of shared memory bank conflicts.

To the best of our knowledge, our work provides the first symbolic approach allowing to formally reason about feasibility of shared memory bank conflicts and the entailed shared memory transactions.

2. Background

We introduce in the following the execution model for programs running on CUDA [12] enabled *Graphical Processing Units* (GPUs),[1] describe the adopted symbolic execution model for such programs and briefly introduce how our work builds on solvers for SMT (satisfiability modulo theories) problems.

Execution model and shared memory. A *kernel* is a routine meant to execute on a GPU by a specified number of threads. Each thread is

[1] Our models could be equally applicable for OpenCL applications.

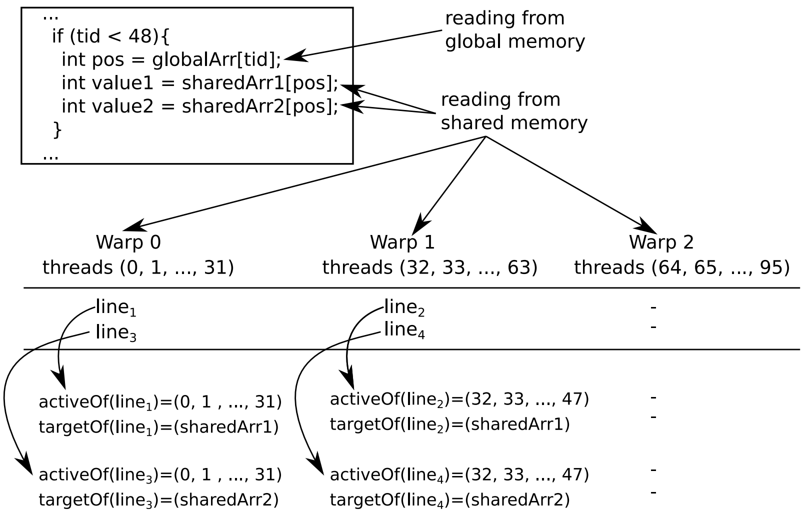


Fig. 1. Example on how `lines` are extracted from the symbolic execution of a kernel code. We assume that three active warps, each with 32 threads, execute the code snippet shown. The third warp, `Warp 2`, does not enter the branch statement, therefore no lines are extracted for it.

identified using a unique identifier. Threads are grouped into *thread blocks*. There is an upper bound (usually 1024) on the maximum number of threads in each thread block. In the same thread block, the threads are further grouped into *warps*. Threads in a single warp execute each instruction in lockstep. Threads in a warp may *diverge* in case they follow different branches of a condition. In such a case, only a subset of the threads inside a warp may be active.

A GPU contains several *streaming multiprocessors*. Each streaming multiprocessor contains multiple *computation cores*. Each computation core can run a single warp at a time. Every streaming multiprocessor comes with a scratchpad memory (more than 48 KiBs on current platforms) that can be shared among the threads running on the streaming multiprocessor. This scratchpad memory is referred to as *shared memory* in CUDA [12] enabled GPUs.[2] Access to the shared memory is orders of magnitude faster than the access to the global memory of the GPU. The shared memory is controlled by the user. When starting a kernel, the user needs to specify how much shared memory the kernel will need or declare all the shared memory variables as static.

Each thread in a thread block is guaranteed to run on the same streaming multiprocessor and has access to the same shared memory. Such a thread can only synchronize with threads in the same thread block. Based on the requirements of shared memory, registers and synchronization barriers, the GPU scheduler is free to schedule multiple thread blocks on the same streaming multiprocessor. The actual limits, such as the number of threads per thread block or the number of streaming multiprocessors and computing cores, are dependent on the compute capability of the underlying GPU.

Symbolic execution for GPU programs. Conceptually, symbolic execution assumes symbolic values for program inputs and collects constraints on such values along the explored paths yielding a *symbolic trace* for each path. The symbolic trace accounts for encountered branches and updates along the explored path. It adds conditions on existing variables (for encountered branches) and on relations between new and existing variables' values (for encountered updates). Obtained constraints can then be solved by a suitable Satisfiability Modulo Theories (SMT) solver to obtain a concrete *model* (see later in this section) from which it is easy to extract a concrete program input that triggers the considered path.

We focus in this work on analyzing warp accesses to shared memory in GPU programs. For this, we build on existing works on symbolic execution for GPU programs [13]. Such works assume a data-race-free GPU program as input and can give, for each program path, a symbolic trace where conditions and updates corresponding to each warp instruction can be captured in terms of symbolic variables. Given such a symbolic trace, we focus on the shared memory accesses and regard such a trace as a sequence of *"lines"*: $\texttt{line}_1, \texttt{line}_2, \ldots, \texttt{line}_n$ where each $\texttt{line}_i$ reflects accesses to shared memory performed during a memory instruction in lockstep by some threads of a warp. Each line, $\texttt{line}_i$, is effectively a list of addresses (symbolic or concrete) that each active thread of the warp accesses through the memory instruction (i.e., $\texttt{line}_i = (addr_i^0, addr_i^1, \ldots, addr_i^{m-1})$) We write $\texttt{activeOf}(\texttt{line}_i)$ to mean the set of thread identifiers that perform the shared memory access at $\texttt{line}_i$ and $\texttt{targetOf}(\texttt{line}_i)$ to mean the array in shared memory that might be accessed at $\texttt{line}_i$. Due to warp divergence, this set of threads can be a strict subset of the total set of threads of the warp (warps are typically 32 threads large) that performed the memory instruction. Given a line $\texttt{line}_i$ and a thread identifier `tid` in $\texttt{activeOf}(\texttt{line}_i)$, we write $\texttt{line}_i[\texttt{tid}]$ to mean the value of the shared memory address accessed by the thread `tid`. This value is in $\texttt{targetOf}(\texttt{line}_i)$ by definition. Symbolically executing the paths means values and addresses may be expressed symbolically in terms of the inputs to the kernel.

Fig. 1 describes how extracted `lines` are related to the shared memory accesses during execution of a kernel code. In this work, we only associate lines to shared memory accesses. We assume three warps, each with 32 threads, will execute the code snippet in the figure. We can see from Fig. 1 that the third warp, `Warp 2`, does not execute the branch instructions due to having the threads with an ID higher than 47. For the same reason, only half the threads of the second warp, `Warp 1`, are active during the execution of the branch statements. Furthermore, in Fig. 1, we can see the concrete values that $\texttt{activeOf}(\texttt{line}_i)$ and $\texttt{targetOf}(\texttt{line}_i)$ contain.

Solvers for SMT (satisfiability modulo theories) problems. SMT solvers can check the satisfiability of first-order logic formulas involving predicate and function symbols that are to be interpreted in some background theories [36]. Examples of background theories include linear integer arithmetic, linear real arithmetic, bitvectors, uninterpreted functions, arrays or strings. The formulas are typically quantifier-free. Recent advances resulted in more powerful and more scalable solvers that became key enablers in several research and industrial applications, including testing, verification, scheduling and artificial intelligence. In this work, SMT solvers are used to solve quantifier-free constraints involving uninterpreted functions, bitvectors and arrays

[2] Called local memory in OpenCL.

```
(set-logic QF_AUFBV)
(declare-fun input () (Array (_ BitVec 8) (_ BitVec 8)))
(define-fun val0 () (_ BitVec 8) (select input (_ bv0 8)))
(define-fun val1 () (_ BitVec 8) (select input (_ bv1 8)))
(assert (bvslt val0 val1))
(define-fun delta () (_ BitVec 8) (bvsub val0 val1))
(assert (= (_ bv0 8) (bvsmod  delta (_ bv32 8))))
(check-sat)
(get-model)
```

Fig. 2. A simple constraint involving bitvectors and a bitvector array.

(i.e., the QF_AUFBVA combination of background theories). For instance, Fig. 2 defines a constraint in the SMT2LIB [37] format. The constraint involves a byte-indexed array `input` of bytes and two values `val0` and `val1` respectively representing the first and the second bytes of the array `input`. The constraint requires that `val0` is strictly smaller than `val1` and that their difference is a multiple of 32. Such constraints can arise, for instance, during symbolic execution where a path requires that `val0` is strictly smaller than `val1`, or when checking for the possibility that the values can be used as indices for bank conflicting shared memory accesses. An SMT solver that handles QF_AUFBV formulas can answer whether such a constraint is satisfiable or not. If it is, the solver can return a *model*, i.e., a valuation that satisfies the constraint and that associates a value to each free variable. An off-the-shelf SMT solver can show the above constraint is indeed satisfiable and might give a model where the first byte of `input` is, for example, 35 and the second one is 67.

3. Identifying bank conflicts in a simple example

We use the simple kernel program of Listing 1 to illustrate the problem we solve in this work and how we tackle it. The goal of the work is to exhibit, or to refute, the existence of an input to a kernel program that results in a given number of shared memory transactions induced by the resulting bank conflicts. A more detailed description of our symbolic approach can be found in Section 4.

A simple example. To simplify the discussion, assume the size `SIZE` of each of the three argument arrays `vector`, `filter` and `output` is 1024. In addition, assume all values in the `vector` array are in the range $[0 \ldots (\mathtt{SIZE}-1)]$ and that the kernel is launched with as many threads as the size of the input arrays (i.e., `SIZE`). These assumptions are solely adopted in this section for simplification purposes. With 1024 threads in a thread block and 32 threads per warp, the execution of the program results in 32 warps. These threads fit in a single thread block and can access the same shared memory (see Section 2). The number of threads per block and per warp is a consequence of the underlying architecture and of the compute capability and our analysis accounts for them.

```
1 #define SIZE 1024
2
3 __global__ void simple(int* vector, int* filter, int* output){
4   __shared__ int shared_filter[SIZE];
5   int tid = gridDim.x * blockIdx.x + threadIdx.x;
6   shared_filter[tid]= filter[tid];
7   __syncthreads();
8   out[tid] = shared_filter[vector[tid]];
9 }
```

Listing 1: Simple kernel with a fixed (row 6) and an input-dependent (row 8) access pattern.

First, each thread computes its thread identifier at row 5. Then, all threads in the block copy the values from the `filter` array in global memory to the `shared_filter` array in shared memory (row 6) and wait for each other at the barrier on row 7. The order in which the 32 warps (with 32 threads each) perform the read from global memory and the write to shared memory (row 6) is determined by the scheduler. However, this order does not matter with regard to the bank conflicting accesses since the program is data-race free. Indeed, data-race freeness ensures that obtained values and addresses do not depend on the scheduling of the warps. Occurrences of bank conflicting accesses, and hence required numbers of shared memory transactions, are determined by the values of the shared memory addresses. Since these addresses are independent of the scheduler, they can be computed for any scheduler. After the barrier, each thread fetches the value `vector[tid]` corresponding to its thread identifier `tid`. It then uses this value to read from the `shared_filter` array and writes the result to the `output` array. The execution of this kernel will therefore result in 32 writes per warp at row 6 to shared memory (each write is performed by one of the 32-threads warps) before the barrier at row 7. This is followed by 32 additional accesses (reads at row 8) from shared memory (each read is performed by one of the 32-threads warps) after the barrier. We will use *line* to denote a shared memory access (read or write) performed in lockstep by a warp (see Section 2). In this kernel program, all threads of a warp are active during each access (this would not have been the case if branches would have been involved). Each line therefore involves 32 shared memory reads or writes performed in lockstep. From a shared memory perspective, the program boils down to 64 lines: 32 writes followed by 32 reads.

Shared memory banks and resulting conflicts. Shared memory can be viewed as an array of SHARED_SIZE bytes indexed from 0 to SHARED_SIZE − 1. Shared memory is partitioned into a number of equally-sized memory modules called banks, typically 32 banks as assumed in this work. Banks are non-contiguous regions of shared memory. Each $\mathtt{bank}_k$ (for $k : 0 \leq k < 32$) handles accesses for the set of all 4-contiguous-bytes words (hereafter *word*) starting at some byte index $\{j | \; 0 \leq j < \mathtt{SHARED\_SIZE} \text{ and } (j/4)\%32 = k\}$. The kernel in the example of Listing 1 accesses the array `shared_filter` consisting in 1024 words of 4 bytes each. This array has therefore 32 words in each one of the 32 banks.

When a thread in some warp requests access to a shared memory location, the request is handled by the bank to which the location belongs. A `bank conflict` appears when two or more threads belonging to the same warp simultaneously request access to different shared memory locations in the same bank. Conflicting memory accesses cannot be handled in the same *memory transaction*. Recall that all active threads in a warp need to finish the current instruction before progressing to the next one due to lockstep execution. If active threads in a warp simultaneously request shared memory locations handled by different banks, then all requests can be handled in parallel by all the different banks. Only one *memory transaction* involving all banks is needed. However, if some active threads in the warp execute an instruction that results in pairwise bank-conflicting accesses (i.e., accesses to distinct shared memory locations in the same bank), then the requests have to be handled sequentially by the corresponding bank in as many memory transactions as the number of accesses.

As a result, requests to different banks result in parallel and efficient memory transactions. Requests to different locations in the same bank result in sequentialized and slower memory transactions. The non-contiguous organization of shared memory banks is useful when threads in the same warp access contiguous words in the shared memory. For instance, accesses by each warp at row 6 in Listing 1 do not result in bank conflicts. They can be efficiently parallelized because the targeted locations in shared memory belong to different banks. They result in one memory transaction per warp executing the instruction. Other access patterns can however result in bank conflicts and in degraded performance. For instance, accesses to shared memory at row 8 depend on the values in the input array `vector`. Some inputs may result in no bank conflicts, while others may result in accesses, by the same warp on the same instruction, to 32 different words of a bank. Such an access would result in a sequentialization of the 32 accesses. Such sequentialization would require 32 serialized memory accesses that will lead to the worst case execution time for the shared memory instruction.

Our approach, as described later in this paper, can provide *concrete* inputs that exhibit the worst case execution time for the shared memory instruction. We show in Section 5 how our approach can aid a programmer in analyzing the worst case memory behavior of a program.

It is also important to note that there are fewer possible combinations of thread accesses to the shared memory when an instruction exhibits the minimum or the maximum number of shared memory transactions. This knowledge can be used as a side channel by an attacker to reduce the search space when trying to deduce secret information. We will show in Section 6 how our approach can quantify the leakage of such a side channel if an attacker observes the maximum number of shared memory transactions during the execution of a cryptographic GPU program.

4. Symbolic identification of bank conflicts

Given a kernel, this work aims to (i) check feasibility of a given number of shared memory transactions due to bank conflicts and (ii) exhibit an input in case this number is feasible. For this, we leverage on symbolic execution in order to encode executions and bank conflicts. Recall from Section 2 that symbolic execution of a path in a GPU program results in a sequence $\mathtt{line}_1, \mathtt{line}_2, \ldots, \mathtt{line}_n$ of shared memory accesses. We assume such a sequence in the following and describe how we check feasibility of a number of memory transactions for the whole program. We will describe three different approaches in this section.

The first approach (Section 4.1) delegates the search to the underlying SMT solver. The idea is to introduce an integer variable for each shared memory word in order to capture whether the word is accessed in a line or not. Constraints are then added to relate these variables to bank conflicts by stating whether simultaneous accesses by some warp to different words belonging to the same bank are possible. The number of conflicts in the same bank captures the number of required shared memory transactions for that bank. The total number transactions to the shared memory is then the maximum number of required transactions for each bank. This encoding requires a number of variables that is linear in the size of the shared memory and delegates the search of possible inputs to the underlying SMT solver.

In our second approach (Section 4.2), the idea is to avoid enumerating the shared memory words and instead to encode possible conflicts between threads of the same warp, regardless of the actual words on which they conflict. Such conflicts can be used to define *"cliques"*, used here to mean threads belonging to the same warp and resulting in pairwise conflicting memory accesses in a line. This allows for a clique-based exploration of possible inputs which allows for a higher level of abstraction.

Since the number of such cliques explodes combinatorically, we finally introduce our lazy approach (Section 4.3) to enumerate cliques on-demand. Reasoning at the level of cliques allows us to cache prior results and to guide the search using conflicts between thread accesses.

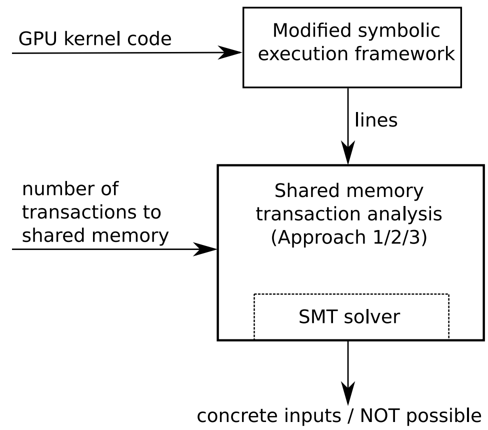


Fig. 3. Overview of the analysis framework.

Fig. 3 gives an overview of our proposed analysis framework.

The modified symbolic execution framework in Fig. 3 is based on GKLEE [13]. We have modified GKLEE to capture all the shared memory accesses during the execution of the GPU kernel. Using this information we can build our lines. We provide the lines and targeted number of transactions to the shared memory to our selected approach in the shared memory transaction analysis. We can then produce concrete inputs that exhibit the targeted number of transactions or establish that such inputs do not exist. The approaches leverage on the underlying SMT solver to achieve this goal. The approaches differ in the queries and in how much they delegate to the solver. In the next subsections we present, in detail, our three approaches.

4.1. Direct approach

This approach encodes possible accesses to each shared memory word and uses the encoding to impose constraints on the numbers of different accessed words in the same bank. For each 4-bytes word beginning at byte address `wadd` in $\mathtt{targetOf}(\mathtt{line}_i)$, we introduce an integer variable $\mathtt{access}^{\mathtt{wadd}}_{\mathtt{line}_i}$. The value of such a variable is 1 if the word starting at `wadd` is indeed accessed by some thread in $\mathtt{line}_i$. The value is 0 otherwise. For each such word in the shared memory, the following constraints are added to the encoding:

$$\left(\bigvee_{\mathtt{tid} \in \mathtt{activeOf}(\mathtt{line}_i)} (\mathtt{line}_i[\mathtt{tid}] = \mathtt{wadd})\right) \Rightarrow (\mathtt{access}^{\mathtt{wadd}}_{\mathtt{line}_i} = 1) \tag{1}$$

$$\left(\bigwedge_{\mathtt{tid} \in \mathtt{activeOf}(\mathtt{line}_i)} (\mathtt{line}_i[\mathtt{tid}] \neq \mathtt{wadd})\right) \Rightarrow (\mathtt{access}^{\mathtt{wadd}}_{\mathtt{line}_i} = 0) \tag{2}$$

For each $\mathtt{bank} : 0 \leq \mathtt{bank} < 32$, we introduce an integer variable $\mathtt{conflict}^{\mathtt{bank}}_{\mathtt{line}_i}$. The value of this variable, for the given `bank` and $\mathtt{line}_i$, is equal to the number of all simultaneous requests to the different words in the shared memory. For a line, $\mathtt{line}_i$, we will have 32 such variables, one for each bank.

The following constraint is therefore added to the encoding:

$$\mathtt{conflict}^{\mathtt{bank}}_{\mathtt{line}_i} = \sum_{\substack{\mathtt{wadd}\%32 = \mathtt{bank} \\ \mathtt{waddintargetOf}(\mathtt{line}_i)}} \mathtt{access}^{\mathtt{wadd}}_{\mathtt{line}_i} \tag{3}$$

The number of transactions $\mathtt{trx}_{\mathtt{line}_i}$ required by $\mathtt{line}_i$ is then the maximum conflicts over all banks. The total number of transactions required for the sequence $\mathtt{seq} = \mathtt{line}_1, \mathtt{line}_2, \ldots, \mathtt{line}_n$ is the sum of the required number of transactions for each line:

$$\mathtt{trx}_{\mathtt{line}_i} = \mathtt{Max}\left\{\mathtt{conflict}^{\mathtt{bank}}_{\mathtt{line}_i} \mid k : 0 \leq k < 32\right\} \tag{4}$$

$$\mathtt{total}_{\mathtt{seq}} = \sum_{i:1\leq i\leq n} \mathtt{trx}_{\mathtt{line}_i} \tag{5}$$

Checking whether the program can generate `target` memory transactions amounts to adding the constraints $\mathtt{trx}_{\mathtt{line}_i} = \mathtt{target}$ and querying the underlying SMT solver for satisfiability. This approach requires one query to the SMT solver during the shared memory transaction analysis step from Fig. 3.

4.2. Conflict-guided approach

For the approach presented in the previous section (see Section 4.1), the size of the formulas depends on the size of the shared memory that might be accessed. We aim to remove this dependency to shared memory size with this current approach. However, this approach will present other limitations that will be mitigated by the approach presented in Section 4.3. This approach can also be used in the shared memory transaction analysis step from Fig. 3. Instead of focusing on whether each word in the shared memory is accessed, we focus on whether active threads in a warp perform bank conflicting accesses. This approach is guided by sets of pairwise conflicting threads belonging to the same warp. We call these *cliques*. Informally, a clique C for a line $\mathtt{line}_i$ is a set of active threads performing pairwise bank conflicting memory accesses in $\mathtt{line}_i$. Given a symbolic sequence $\mathtt{line}_1, \mathtt{line}_2, \ldots, \mathtt{line}_n$, the idea is to find possible maximal cliques for each line. If C is a clique for $\mathtt{line}_i$, then the line will result in at least $|C|$ memory transactions. If there are no cliques of some size c for $\mathtt{line}_i$, then it cannot result in c or more memory transactions.

Given a line $\mathtt{line}_i$ and a set C of threads in $\mathtt{activeOf}(\mathtt{line}_i)$, the constraint $\mathtt{mkf}(C, \mathtt{line}_i)$ defined in Eq. (6) enforces that the shared memory accesses performed by the threads in C result in pairwise bank conflicts, and hence in at least $|C|$ memory transactions.

$$\mathtt{mkf}(C, \mathtt{line}_i) = \bigwedge_{\substack{\mathtt{tid}_1 \neq \mathtt{tid}_2 \\ \mathtt{tid}_1, \mathtt{tid}_2 \in C}} \begin{pmatrix} \mathtt{line}_i[\mathtt{tid}_1] \neq \mathtt{line}_i[\mathtt{tid}_2] & \wedge \\ (\mathtt{line}_i[\mathtt{tid}_1] - \mathtt{line}_i[\mathtt{tid}_2])\%32 = 0 \end{pmatrix} \tag{6}$$

If $\mathtt{mkf}(C, \mathtt{line}_i)$ is satisfiable, we say the clique C is *enforceable* for $\mathtt{line}_i$. Otherwise, C is said to *not be enforceable*. A line $\mathtt{line}_i$ where exactly n_i shared memory transactions are possible has at least one enforceable clique of size n_i and no enforceable clique with a strictly larger size. We use this in Algorithm 1. The algorithm takes a `target` number of transactions and a symbolic sequence $seq = \mathtt{line}_1, \mathtt{line}_2, \ldots, \mathtt{line}_n$ of n warp accesses to shared memory. The algorithm uses cliques to guide the search for possible inputs that result in `target` memory transactions. It instantiates a *generator* Γ at row 1. In addition to the target number of transactions, `target`, the generator takes as argument the set of threads active at each line. Intuitively, the generator will be used to propose sequences of cliques of the form $\langle C_1, \ldots, C_n \rangle$ where (i) the set of threads C_i is a subset of $\mathtt{activeOf}(\mathtt{line}_i)$, and (ii) the sum $\sum_{i:1\leq i\leq n} |C_i|$ coincides with the target number `target` of shared memory transactions. The generator is used to produce, at row 3, a sequence of n candidate maximal cliques. One clique for each line. This could for instance be implemented using iterators. In row 5, the algorithm encodes, for each $\mathtt{line}_i$ with a candidate clique C_i, that there is at least a clique C with a cardinality that is strictly larger than $|C_i|$. The SMT solver is called at row 6. It checks the satisfiability of the constraint stating that for each line $\mathtt{line}_i$, the clique C_i is enforced but no clique with more than $|C_i|$ threads can be enforced. This amounts, for each line $\mathtt{line}_i$, to enforcing that C_i is a maximal set of threads with pairwise bank conflicting shared memory accesses at line $\mathtt{line}_i$. The underlying SMT solver returns a model `m` if the constraint is satisfiable. The model is a valuation of all free variables in the checked constraint. A concrete input (i.e., a test) can be easily extracted from it. Such an input will satisfy the constraint and will therefore result, for each line $\mathtt{line}_i$, in $|C_i|$ shared memory transactions. A concrete input is therefore returned at row 7. Otherwise, the constraint is not satisfiable and the produced sequence of cliques is excluded at row 8 before a new iteration of the algorithm.

input : a `target` number of transactions and a symbolic sequence of n lines $seq = \mathtt{line}_1, \mathtt{line}_2, \ldots, \mathtt{line}_n$.
output: not_feasible or a concrete input with `target` transactions.

1 $\Gamma \leftarrow Gen(\langle \mathtt{activeOf}(\mathtt{line}_1), \ldots, \mathtt{activeOf}(\mathtt{line}_n) \rangle, \mathtt{target})$;
2 **while** $(\neg(\Gamma.\mathtt{is\_empty}()))$ **do**
3 $\quad \langle C_1, \ldots, C_n \rangle \leftarrow \Gamma.\mathtt{get\_candidate}()$;
4 $\quad$ **foreach** $i \leftarrow 1$ *to* n **do**
5 $\qquad \mathtt{at\_most}_i \leftarrow \bigvee_{\substack{C \subseteq \mathtt{activeOf}(\mathtt{line}_i) \\ |C| = |C_i|+1}} \mathtt{mkf}(C, \mathtt{line}_i)$;
6 $\quad \mathtt{sat}, \mathtt{m} \leftarrow \mathtt{check}\left(\wedge_{1\leq i\leq n} \mathtt{mkf}(C_i, \mathtt{line}_i) \wedge \neg\vee_{1\leq i\leq n}\ \mathtt{at\_most}_i\right)$;
7 $\quad$ **if** `sat` **then return** $\mathtt{init\_of}(\mathtt{m})$;
8 $\quad$ **else** $\Gamma.\mathtt{forbid\_exact}(\langle C_1, \ldots, C_n \rangle)$;
9 **return** not_feasible

Algorithm 1: Conflict-guided exploration

Termination of the algorithm is ensured by the finite number of candidates and the fact that each sequence of candidates is checked at most once. The algorithm has the advantage of avoiding the explicit enumeration of all words in the shared memory. It has also the advantage of directing the search by reasoning at the level of cliques, and not at the level of accessed words. This level of abstraction is not available when relying on the SMT solver directly as in the first approach. An important limitation of this approach is, however, the combinatorial explosion of the number of candidate cliques. For $|\mathtt{activeOf}(\mathtt{line}_i)|$ threads in a $\mathtt{line}_i$ (up to 32 threads), there will be $\binom{|\mathtt{activeOf}(\mathtt{line}_i)|}{\mathtt{candidate\_size}}$ candidate cliques of size `candidate_size`. To become a clique, such a candidate clique C_i requires that its elements have pairwise conflicting accesses to shared memory. To be maximal, there should be no clique with a larger cardinality. For each such candidate clique C_i at row 6, there are $\binom{|\mathtt{activeOf}(\mathtt{line}_i)|}{1+|C_i|}$ subsets that should not form a clique. Instead of eagerly enumerating all such constraints, we propose in the next section to generate them on-demand.

4.3. On-demand conflict-guided approach

The conflict-guided approach from Section 4.2 does not enumerate all shared memory words as described in the direct approach from Section 4.1. It, however, suffers from the combinatorial explosion in the number of cliques to be refuted at each SMT query at row 6 in Algorithm 1. For each candidate clique C_i in a sequence of cliques $\langle C_1, \ldots, C_n \rangle$, there are $\binom{|\mathtt{activeOf}(\mathtt{line}_i)|}{1+|C_i|}$ cliques to be refuted. In this section, we aim to mitigate this limitation by refuting cliques on demand. The idea is to first perform simpler queries one line at a time (i.e., locally to the lines), to cache the answers together with on-demand generated cliques that need to be refuted. This allows us, when performing queries for all the lines together (i.e., globally for the candidate sequence of cliques) to avoid non-enforceable candidate cliques and strictly larger cliques that appeared in previous checks. This third approach can be used in the shared memory transaction analysis step in Fig. 3. Algorithm 2 describes this approach.

The generator used in Algorithm 2 caches more information than the one used in Algorithm 1. For each line $\mathtt{line}_i$, with $i : 1 \leq i \leq n$, the generator can associate a set $\mathcal{F}_i$ of *spoilers* to each candidate clique C_i for the given line. A spoiler $\mathcal{S}$ for C_i at line $\mathtt{line}_i$ is a clique for $\mathtt{line}_i$ with $|\mathcal{S}| = 1 + |C_i|$. The algorithm lazily identifies such spoilers from the concrete models found by the SMT queries. A spoiler for a clique C_i in a model `m` returned by an SMT query refutes the maximality of C_i in `m`. Concretely, it means that the input extracted from `m` would result, for $\mathtt{line}_i$ in strictly more transactions than $|C_i|$. Instead of eagerly forbidding all spoilers as done in rows 5 and 6 of Algorithm 1, this algorithm considers that all candidate cliques for all lines are initially associated to an empty list of spoilers to be forbidden. The first time a candidate clique is proposed in a sequence in row 3, it is associated to an empty set $\mathcal{F}$ of spoilers to be forbidden. Then, it lazily forbids, for each line and each candidate clique, spoilers based on the models returned by the SMT queries at rows 6 and 21 of Algorithm 2. A call to $\texttt{compute\_a\_spoiler}(\texttt{m}, \texttt{line}_i, C_i)$ at rows 8 or 26 finds a spoiler $\mathcal{S}$, if any, for the candidate clique C_i at $\texttt{line}_i$ in $\texttt{m}$. Indeed, all values are concrete in $\texttt{m}$. This includes the shared memory addresses corresponding to the line $\texttt{line}_i$. The procedure $\texttt{compute\_a\_spoiler}(\texttt{m}, \texttt{line}_i, C_i)$ identifies the conflicting pairs for line $\texttt{line}_i$ and can find, if any, some spoiler clique $\mathcal{S}$. It returns the empty set if there are no such spoilers. A found spoiler $\mathcal{S}$ is associated to clique C_i for $\texttt{line}_i$ with existing spoilers $\mathcal{F}_i$, using $\Gamma.\texttt{update\_spoiler}(\{\mathcal{S}\} \cup \mathcal{F}_i, \texttt{line}_i, C_i)$ as done in rows 13 and 28 of Algorithm 2. For each candidate sequence with its cached spoilers $\langle(C_1, \mathcal{S}_1), \ldots, (C_n, \mathcal{S}_n)\rangle$, the algorithm starts by issuing an SMT query for each line on its own and caching the obtained results (rows 5–19). Based on the cached information, it may issue an SMT query for all lines (rows 21–32). We describe these steps in the following.

```
   input : a target number of transactions and a symbolic sequence of
           n lines seq = line_1, line_2, ..., line_n.
   output: not_feasible or a concrete input with target transactions.
1  Γ ← Gen(⟨activeOf(line_1), ..., activeOf(line_n)⟩, target);
2  while (¬(Γ.is_empty())) do
3      ⟨(C_1, F_1), ..., (C_n, F_n)⟩ ← Γ.get_candidate();
4      new_choice ← true ;
5      foreach i ← 1 to n do
6          sat, m ← check( mkf(C_i, line_i) ∧ ¬(⋁_{S∈F_i} mkf(S, line_i)) );
7          if sat then
8              S ← compute_a_spoiler(m, line_i, |C_i|);
9              if S.is_empty() then
10                 new_choice ← false;
11                 continue;
12             new_choice ← true ;
13             Γ.update_spoiler({S} ∪ F_i, line_i, C_i);
14         else
15             if F_i.is_empty() then
16                 Γ.forbid_supersets(line_i, C_i);
17             else
18                 Γ.forbid_exact(line_i, C_i)
19         break;
20     if new_choice then continue;
21     sat, m ← check(∧_{1≤i≤n} (mkf(C_i, line_i) ∧ ¬∨_{S∈F_i} mkf(S, line_i)));
22     if (¬sat) then
23         Γ.forbid_exact(⟨C_1, ..., C_n⟩);
24     else
25         foreach i ← 1 to n do
26             S ← compute_a_spoiler(m, line_i, |C_i|);
27             if S.is_empty() then continue;
28             Γ.update_spoiler({S} ∪ F_i, line_i, C_i);
29             new_choice ← true ;
30             break;
31         if new_choice then continue;
32         return init_of(m)
33 return not_feasible
```

Algorithm 2: On-demand conflict-guided exploration

The first step has a *local* scope for each line. In the first step each $\texttt{line}_i$ is checked separately (rows 5–19 in Algorithm 2) without adding constraints for the other lines in the sequence. If the candidate clique C_i is enforceable, i.e., we obtain a model $\texttt{m}$ from the SMT solver (row 6 in Algorithm 2). In this case, we know there is at least an input resulting in at least $|C_i|$ transactions.

If the concrete input results in exactly $|C|$ transactions (empty spoilers at row 9 in Algorithm 2), then we found a solution for $\texttt{line}_i$ and we can proceed to checking the other lines separately. If the concrete input in $\texttt{m}$ results in at least $1 + |C_i|$ transactions, then we can find and forbid a corresponding spoiler $\mathcal{S}$ to symbolically eliminate a whole family of inputs resulting in the same spoiler. Observe the particular model $\texttt{m}$ is also eliminated. If the candidate clique C_i is not enforceable for $\texttt{line}_i$ even without any enforced spoiler, then we can avoid issuing queries involving supersets of C_i for $\texttt{line}_i$ as the constraints imposed by C_i were already not satisfiable (row 16 in Algorithm 2). Repeating the process of forbidding spoilers based on the obtained models allows to lazily guide the search for a given line (hence simpler queries) while eliminating families of inputs, one for each found spoiler. This lazy approach will either find a concrete input resulting in exactly n_i transactions (i.e., no spoiler) or will stop obtaining solutions. In the latter case we can conclude that it is impossible to obtain exactly $|C_i|$ transactions when enforcing C_i for $\texttt{line}_i$ (row 18 in Algorithm 2).

The second step has a *global* scope for the whole sequence of lines. In the second step (rows 21–32 in Algorithm 2), once we have obtained the locally enforceable cliques (and the corresponding spoilers to be forbidden) that can result in exactly n_i transactions for each line $\texttt{line}_i$, we can check whether the constraints obtained for each $\texttt{line}_i$ can be satisfied together (row 21 in Algorithm 2). Indeed, there might be no common input that would allow to enforce the candidate cliques. If there is no model, then we need to proceed with other choices of cliques after forbidding the current sequence of clique candidates (rows 22–23 in Algorithm 2). If the query has a model $\texttt{m}$, then we check whether the number of transactions for each $\texttt{line}_i$ is still n_i. We can again forbid spoilers and restart with a new candidate sequence (rows 25–31 in Algorithm 2). If there are no spoilers, then we found a model where the lines give exactly $\texttt{target}$ shared memory transactions. We extract an input from the obtained model and return it (row 32 in Algorithm 2). This is repeated until a concrete input is found or we run out of possible candidate sequences.

Termination of Algorithm 2 is guaranteed by observing that a sequence of candidates and spoilers is proposed at most once, and that there are finitely many candidates and spoilers. Correctness is guaranteed by observing that we only exclude spoilers or impossible candidates. This approach tries to avoid, for a given sequence of candidate cliques, to eagerly enumerate the intractable number of spoilers (like in the eager approach from Section 4.2) while explicitly manipulating cliques to identify the accesses responsible for the targeted number of transactions (something the direct approach from Section 4.1 misses).

4.4. Direct approach vs. On-demand conflict-guided approach

We used a modified version of Listing 1 as a benchmark to compare the direct and the on-demand conflict-guided approaches in terms of analysis time. The modifications differ in the number of lines and the size of the shared memory. Listing 1 adopts 1024 integer (i.e., 4096 bytes) and has one row reading shared memory with an input dependent pattern. It corresponds to the first entry of Table 1. By adding another instruction to read from shared memory (like in row 8) we obtain a kernel with two *lines*. The performance results for the kernels with two and three *lines* can also be seen in Table 1.

Due to the combinatorial explosion explained in Section 4.2, the conflict-guided approach was much slower compared to the other two approaches and, therefore, was not included in the comparison in Table 1.

The results in Table 1 show that the execution time for the on-demand conflict-guided approach does not depend on the size of the shared memory, but only on the number of lines being analyzed.[3] The direct approach requires, as expected, increasingly more analysis time the more shared memory is being used. Considering these results, we have decided to use the on-demand conflict-guided approach in our application to performance (Section 5) and security analysis (Section 6).

[3] There are relatively minor disturbances in the execution times for a given number of lines, due to factors such as: the SMT solver internal workings, other active processes, the operating system, etc.

Table 1
Comparing analysis time of the direct approach vs. the on-demand conflict-guided approach for the maximum number of possible shared memory transactions of one active warp.

Number of lines	Shared memory size (bytes)	Direct approach (seconds)	On-demand conflict-guided approach (seconds)
1	4096	87.59	4.7
	8192	300.9	4.6
	16 384	1059.21	4.5
2	4096	346.3	14.4
	8192	537.48	12.9
	16 384	4242.47	19.9
3	4096	453.98	27.39
	8192	1010.21	25.7
	16 384	5823.45	27.6

Improving analysis time. We have also implemented a heuristic to speed-up the shared memory transaction analysis part from Fig. 3. It corresponds to a preprocessing step that aims to reduce the number of lines that need to be analyzed. The idea is to avoid analyzing two lines if, no matter the input, the numbers of transactions required by each one of them are always equal. We call such lines duplicates. This preprocessing step leverages on the SMT solver for queries to recognize duplicate lines (in terms of shared memory bank conflicts)

We consider that a line $\mathtt{line}_j$ is a duplicate of line $\mathtt{line}_i$ if no matter the input, $\mathtt{line}_j$ always produces the same number of shared memory transactions as $\mathtt{line}_i$. A sufficient condition to check whether $\mathtt{line}_i$ and $\mathtt{line}_j$ are duplicates is to test whether both lines have the same set of active threads, and that the accessed memory addresses in $\mathtt{line}_i$ can be obtained by shifting those in $\mathtt{line}_j$. Intuitively, this guarantees that accesses by a pair of threads in $\mathtt{line}_i$ result in a bank conflict iff accesses by the same pair, and for the same input, also conflict in $\mathtt{line}_j$. Recall that $\mathtt{line}_i[\mathtt{tid}_k]$ represents the value of the shared memory address that the thread $\mathtt{tid}_k$ accesses at line $\mathtt{line}_i$. Practically, and assuming $\mathtt{tid}_0 \in \mathtt{activeOf}(\mathtt{line}_i)$ with $\mathtt{activeOf}(\mathtt{line}_i) = \mathtt{activeOf}(\mathtt{line}_j)$, we conclude that lines $\mathtt{line}_i$ and $\mathtt{line}_j$ are duplicates if the following disjunction is unsatisfiable:

$$\bigvee_{\mathtt{tid} \in \mathtt{activeOf}(\mathtt{line}_i)} \left(\mathtt{line}_i[\mathtt{tid}_0] - \mathtt{line}_j[\mathtt{tid}_0] \neq \mathtt{line}_i[\mathtt{tid}] - \mathtt{line}_j[\mathtt{tid}]\right) \quad (7)$$

5. Application to performance analysis

Analyzing performance and latency of GPU programs is relevant for most GPU applications. They are crucial for real-time systems and control applications where correctness and stability depend on them. Several works have shown the direct impact shared memory bank conflicts have on performance and latency in GPU programs [1–3]. Estimation can proceed by testing as many concrete inputs as possible. Importantly, this only gives a partial picture for programs where the access patterns to shared memory depend on the input values. Testing, therefore, only explores a tiny portion of the possible behaviors. Instead, our approach formally represents constraints capturing the shared memory transactions due to bank conflicts. This allows us to systematically explore the best and worst possible performances based on the feasible numbers of shared memory transactions.

This section is organized in two parts. In the first part, we use a heatmap example to describe how one can use our framework to identify concrete inputs for given numbers of shared memory transactions. Then, we experimentally show that the obtained concrete inputs indeed impact the execution time. In the second part, we compare the worst case performance supplied by our on-demand conflict-guided approach and the one supplied by `GDivAn` [35], a recent tool for deriving worst case execution time for GPU kernels.

5.1. Deriving extreme numbers of shared memory transactions

Our on-demand conflict-guided approach is useful for finding concrete inputs that lead to different numbers of shared memory transactions. Typically, handling larger input sizes in GPU implementations (e.g., corresponding to richer input details or resolutions) entails the usage of larger portions of the shared memory. However, the amount of shared memory a program uses does impact the possibility of bank conflicts. Indeed, access to larger shared memory means more accesses might conflict on the same banks. We use our framework to exhibit, for a program performing the same amount of work but for different shared memory sizes reflecting different levels of handled details, concrete inputs resulting in minimal and maximal numbers of shared memory transactions. We then use these inputs in a real implementation to show how this impacts execution time and performance. We argue that such applications of our framework can allow to strike different tradeoffs between shared memory usage (i.e., potential level of detail) and the required numbers of memory transactions (i.e., expected performance impact).

We adopt the heatmap kernel of Listing 2 to test our on-demand conflict-guided approach and to derive concrete `input` values leading to maximal and minimal numbers of shared memory transactions for a warp. This kernel is representative of GPU programs where shared memory access patterns do depend on input values. We then use the obtained concrete inputs in executions on an actual GPU with 2^{20} threads and evaluate their impact on performance. The heatmap kernel uses the values stored in an array `input` to compute a coloring scheme for three output colors based on red, green and blue filters. The input filters are written to shared memory (rows 14–19) and the shared-memory-saved filters are read in rows 24–34. The values passed in the `input` array dictate the access patterns applied when the filters in shared memory are read (rows 24–34). The size of the `input` and `output` arrays are equal to the total number of threads the kernel has been started with (i.e., `totalThreads`). By changing the value of `SHARED_ARR_SZ`, we allow for different sizes of the shared memory to be read. This can give the possibility for a warp to simultaneously access different locations in the same memory banks, and hence for different bank-conflicts. Observe that given a size for `SHARED_ARR_SZ` and a number of threads, the way warps write into shared memory (rows 14–19) does not depend on the input values. Instead, we focus on the reading part (rows 24–34) as input values can result in different access patterns.

When applying our on-demand conflict-guided approach, we adopt a warp of 32 threads accessing arrays of constant sizes (both values of `totalThreads` and `SHARED_ARR_SZ` are fixed). At the same time, we allow for the `input` values to be symbolic (hence for unknown and potentially different reading patterns in rows 24–34). We then ask our approach for `input` values leading to the maximal and minimal numbers of shared memory transactions.

We employ our on-demand conflict-guided approach on different values for the `SHARED_ARR_SZ` parameter in Listing 2. We have 32 banks handling the shared memory calls. Each bank controls 4 consecutive bytes at a time. We call these 4 consecutive bytes a *box*. We expect that accesses by a 32-threads-warp to a shared memory with 128 bytes (32 boxes) or less will result in one memory transaction while those to a shared memory with 4096 bytes or more will result in at most 32 transactions. Indeed, for a warp with 32 active threads, if we have a shared memory size of 32×4 bytes or less, then we cannot have more than one transaction since for a conflict to occur, a warp needs to access two different boxes belonging to the same bank. On the other hand, if the accessed shared memory is large enough so that each bank has more than 32 integers (i.e., shared memory is at least $32 \times 32 \times 4$ bytes), then the number of transactions is limited by the number of threads in a warp since there cannot be more accesses than there are threads in a warp.

```
1 #define SHARED_ARR_SZ (32 * 32) //we analyze the impact of this parameter
2                                 //on number of shared memory transactions
3
4 __global__ void heatmapKernel(int* input,
5            int* filterRed, int* filterGreen, int* filterBlue,
6            int* outputRed, int* outputGreen, int* outputBlue,
7            long int totalThreads){
8   __shared__ memType sharedFilterRed[SHARED_ARR_SZ];
9   __shared__ memType sharedFilterGreen[SHARED_ARR_SZ];
10  __shared__ memType sharedFilterBlue[SHARED_ARR_SZ];
11  int gid = gridDim.x * blockIdx.x + threadIdx.x;
12  int pos = threadIdx.x;
13
14  while (pos < SHARED_ARR_SZ){
15    sharedFilterRed[pos] = filterRed[pos];
16    sharedFilterGreen[pos] = filterGreen[pos];
17    sharedFilterBlue[pos] = filterBlue[pos];
18    pos += blockDim.x;
19  }
20
21  __syncthreads();
22
23  while (gid < totalThreads){
24    outputRed[gid]= sharedFilterRed[input[gid] % SHARED_ARR_SZ];
25    outputGreen[gid]= sharedFilterGreen[input[gid] % SHARED_ARR_SZ];
26    outputBlue[gid]= sharedFilterBlue[input[gid] % SHARED_ARR_SZ];
27
28    outputRed[gid]+= sharedFilterRed[(input[gid] + 1) % SHARED_ARR_SZ];
29    outputGreen[gid]+= sharedFilterGreen[(input[gid]+1) % SHARED_ARR_SZ];
30    outputBlue[gid]+= sharedFilterBlue[(input[gid] + 1) % SHARED_ARR_SZ];
31
32    outputRed[gid]+= sharedFilterRed[(input[gid] + 2) % SHARED_ARR_SZ];
33    outputGreen[gid]+= sharedFilterGreen[(input[gid]+2) % SHARED_ARR_SZ];
34    outputBlue[gid]+= sharedFilterBlue[(input[gid] + 2) % SHARED_ARR_SZ];
35
36    outputRed[gid] /= 3;
37    outputGreen[gid] /= 3;
38    outputBlue[gid] /= 3;
39
40    gid += blockDim.x * gridDim.x;
41  }
42 }
```

Listing 2: Heatmap kernel code.

We use our approach for a warp of 32 active threads to produce inputs that lead to the minimum/maximum possible number of shared memory transactions. Our approach established that the minimum is always one transaction per line for the kernel in Listing 2 and that the maximum per line varies with the size of the shared memory. Obtaining the concrete inputs for each `SHARED_ARR_SZ` value, for our on-demand conflict-guided approach, takes less than two minutes. Comparing the execution time, for the obtained inputs, on real hardware for only one warp does not yield measurable results since the difference is in tens of cycles. For our experiments we have considered 2^{20} threads. For this reason, we scale the inputs we have obtained from our approach to be able to run with more than 32 threads (i.e., one warp). For this, we replicate the input array for each new warp added. This means that our initial array of 32 values ($\langle val_0, \dots, val_{31} \rangle$), will be reused when adding a new running warp.

Fig. 4 shows the results of testing the kernel using inputs derived from our on-demand conflict-guided approach and corresponding to minimum and maximum numbers of shared memory transactions. In Fig. 4 we are running with 2^{20} threads (i.e., 2^{15} warps). The experiments with 2^{20} threads from Fig. 4 have been performed on an NVIDIA GeForce RTX 2060 SUPER GPU.[4] The machine used for the tests was running Ubuntu 18.04 (64 bit) with an Intel i7-9700 CPU and 16 GB of RAM. The values on the X axis in Fig. 4 reflect the size of the used shared memory (i.e., the actual values set for `SHARED_ARR_SZ` in Listing 2) adopted in the respective experiment.

We have set the value of $totalThreads$ in the kernel in Listing 2 to be 2^{20} when collecting the results for Fig. 4. This was to guarantee the same number of shared memory read instructions being executed for all test results shown in Fig. 4. This is important since the same read instructions can produce different shared memory transactions. This is not the case for the shared memory write instructions in rows 14, 15, 16 of lines Listing 2 since they always produce one transaction per warp for each line. To make sure that the number of write operations in rows 14, 15, 16 do not alter the results, we have also experimented with writing a fixed amount of values into shared memory for all tests. This gave the same results as the ones depicted in Fig. 4. Therefore, the fixed writing patterns had no significant impact on execution time.

As expected, we can see in Fig. 4 that, for a shared memory size of 128 bytes (32 boxes), the execution times are the same for both the maximum and minimum possible numbers of transactions, since they equal one possible transaction per line. We also note that going above the 4096 bytes (1024 boxes) yields similar execution times to the case when the shared memory is exactly 4096 bytes. The small variance in execution time above 4096 bytes is expected as there are other factors (other processes, cache, etc.) that might impact the execution time. The same argument can be made regarding the minimum possible number of transactions for different shared memory sizes. We expect the execution times to be the same since the number of minimum shared memory transactions are the same (i.e., one per line in this experiment).

As we can see from Fig. 4, our on-demand conflict-guided approach has helped produce inputs that do impact the execution time of a GPU kernel even when the number of executed instructions remains the same. Such inputs can provide a programmer, that needs to meet

[4] Our approach is valid for other modern GPUs that have shared memory.

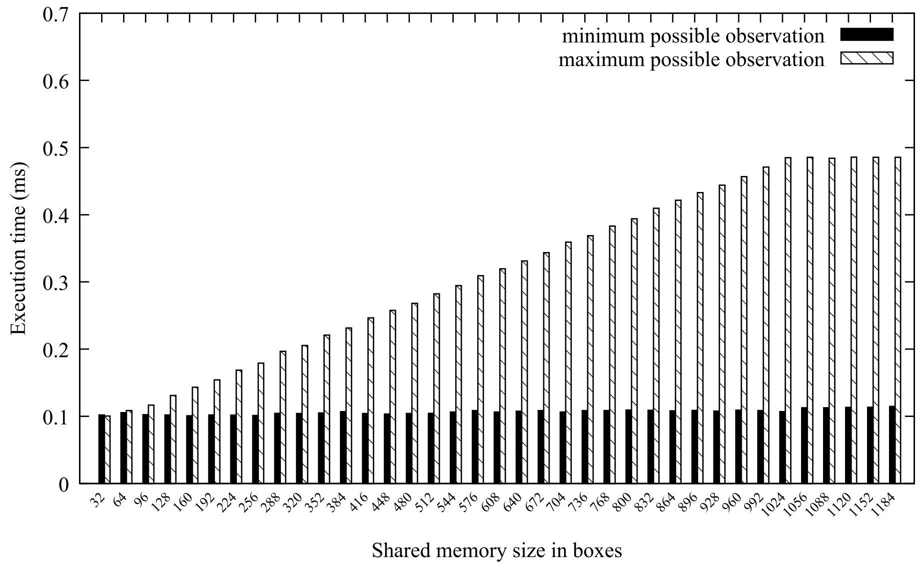


Fig. 4. Execution time for the heatmap kernel with different shared memory sizes (i.e., the value of SHARED_ARR_SZ) and inputs that yield minimum/maximum shared memory transactions. All execution are performed with 2^{20} threads. Each box is 4 consecutive bytes that are handled by the same bank.

execution time deadline requirements, with information on where the trade-off point is when selecting between program performance and output precision. The program's worst case performance is better for a smaller used shared memory in Listing 2 (see Fig. 4). However, selecting a larger shared memory for the filters allows the programmer to have a higher precision filter when representing heatmap colors. This higher precision comes with the cost of a degrading performance, as our results show in Fig. 4. Our approach allows the designer to choose the best quality alternative affordable from the point of view of execution time.

5.2. Additional experiments and comparison of WCET estimations

We have applied the same approach presented in Section 5.1 to detect the worst case performance for two more GPU kernels. One of them is a 256-bin histogram implementation [38]. The other is a convolution kernel in which access to the shared memory depends on the values of the input. Furthermore, we have compared the execution time with respect to inputs derived by our on-demand conflict-guided approach for maximum numbers of shared memory transactions against the execution times produced by a recent WCET estimation tool for GPU kernels. The tool (`GDivAn` [35]) uses a combination of symbolic execution and genetic algorithm to analyze the kernel structure (symbolic execution part) and converge towards the WCET (genetic algorithm part). The results of the comparison are listed in Table 2.

Like for the heatmap kernel procedure from Section 5.1, the on-demand conflict-guided approach is used to find an input with a maximum number of shared memory bank transactions. For this, the code analysis needs to terminate before testing the obtained concrete input on a real GPU. This is unlike the approach adopted by `GDivAn` where new inputs are continuously generated via the genetic algorithm part and tested on a real GPU. Therefore, during the execution of `GDivAn`, the measured WCET might be found earlier than the allocated analysis time. That is the reason the `GDivAn` tool has an extra column in Table 2 which corresponds to when the measured WCET was found. We also note that the `GDivAn` tool can continue to search for a higher WCET if more analysis time is allocated. However, for all the applications in Table 2, after the allocated analysis time (i.e., "Total analysis time"), at least 10 generations (in the genetic algorithm part) have passed with no detected improvement.

We can see from Table 2 that our on-demand conflict-guided approach outperforms `GDivAn` by detecting inputs that lead to execution times that are between 16,9% and 270,8% larger. This is due to the fact that our proposed approach explicitly reasons about hardware related aspects, specifically the number of shared memory transactions.

6. Application to side-channel analysis

Side-channel attacks leverage on observations of micro-architectural behaviors during program executions to leak information about secret inputs, including keys for cryptographic algorithms. Such observations may involve estimating energy consumption, specific cache misses or their numbers in sequential programs, or numbers of shared memory transactions in GPU implementations.

Fig. 5 was obtained by observing the number of generated shared memory transactions for a GPU implementation (provided in [5]) of the cryptographic algorithm AES with a fixed plaintext and for 30 000 randomly chosen 128-bits long keys. The figure represents the distribution of the sampled keys in terms of the number of shared memory transactions they yield. Recall that the AES cryptographic algorithm takes as inputs a secret key and a plaintext to be encrypted and returns the encrypted text. The dependence of the number of shared memory transactions on the input can be explained as follows. AES implementations on GPU devices can store tables in the shared memory of the device. Table values that are actually read from the shared memory during execution depend on the plaintext and on the secret key. The number of shared memory transactions for a fixed plaintext can be observed by an attacker feeding the implementation with the text and measuring the time taken by the execution or by profiling the execution. Fig. 5 illustrates two relevant aspects for side-channel attacks: (i) secret inputs may result in different observations, and (ii) observations can leak different amounts of information. Indeed, an observation that is only possible with few keys will be more valuable to an attacker than an observation that can be obtained from most keys. Extreme cases include situations where all keys give the same observation (no information is leaked as attackers cannot differentiate keys based on the observation) or when an observation is only possible with a specific key (highest possible leakage as attackers can deduce the key just from witnessing the observation).

Attack models. Application of our framework to the analysis of side-channel information leakage assumes the possibility for the attacker to observe the number of shared memory transactions. We assume the attacker can identify the start and the end of a GPGPU routine to

Table 2
Analysis of WCET using GDivAn versus on-demand conflict-guided approach. All experiments were performed on an NVIDIA GeForce RTX 2060 SUPER GPU with an Intel i7 machine having 16 GB RAM and running Ubuntu 18.04.

Program name	GDivAn			On-demand conflict-guided approach		% increase by using the On-demand conflict-guided approach
	WCET (ms)	WCET reached after (s)	Total analysis time (s)	WCET (ms)	Total analysis time (s)	WCET value improvement (%)
Heatmap	0.140	1117	2000	0.482	90	244.2%
Histogram256	1.738	379	1709	2.033	6	16.9%
Convolution	1.083	2188	2339	4.016	316	270.8%

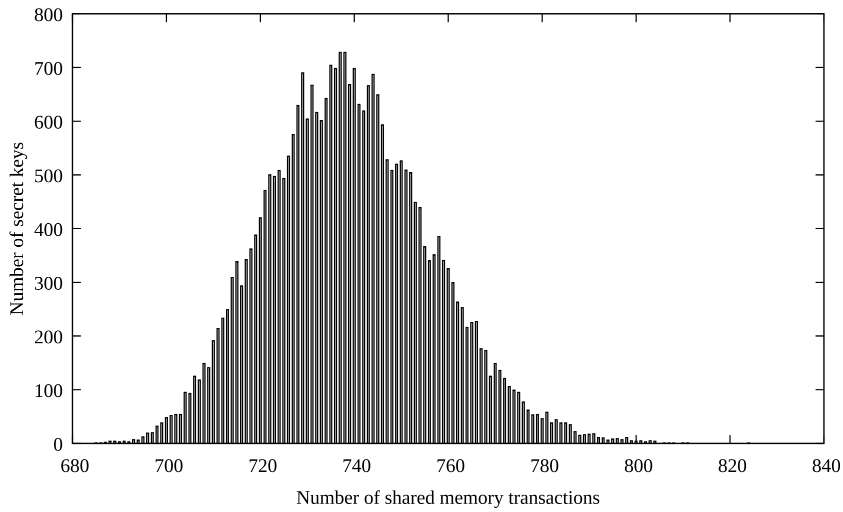


Fig. 5. Example of side-channel observations and their distribution. Plotting the number of keys (30 000 randomly chosen ones) leading to the same number of shared memory transactions for a GPU implementation [5] of AES-128.

accurately attribute the number of shared memory transactions to the routine. This is in line with the synchronous attack model explored in earlier work [39]. Our work also assumes data-race free GPU programs. Without this assumption, actual executions might depend on the underlying scheduler, something we do not have access to. In addition, we do not account for executions of other kernels on the same device or for transfers from global/main memory or for memory leakage outside the GPU (e.g., in preprocessing steps involving the secret information on the host). Executions of other kernels will result in additional shared memory transactions, but these can be analyzed in isolation given there is space for both kernels in the shared memory. We assume such a strong attacker model as it helps us to stress the weaknesses of an application in a favorable situation for the attacker. Still, symbolically reasoning about observable shared memory transactions will be at the core of extensions to this work. For instance, by developing bank-conflict-aware abstractions for preprocessing steps and exposing them to the analysis on the device.

Quantifying extreme leakage. In this section, we target GPU implementations of the AES-128 cryptographic algorithm and adopt as observations the number of shared memory transactions. We assume the plaintext is known to the attacker. We could use our conflict-guided approach to check feasibility of cliques corresponding to minimum and maximum numbers of memory transactions. This would put a bound on the number of possible observations. We choose to go one step further. We conjecture, based on the results from Fig. 5, that few secret inputs result in maximum numbers of shared memory transactions and settle for quantifying information leakage when observing such an extreme number of transactions. Quantifying this leakage is well beyond capabilities of existing works and is made possible because our conflict-guided approach can either provide secret keys that exhibit a given number of shared memory transactions or refute their existence

Table 3
Considered GPU implementations of AES-128.

Implementation	Keys	Min plaintext	Min threads	Rounds (total)	Lines	Unique lines
OpenSSL [6]	16 bytes	16 bytes	4	2 (16)	24	12
ISPASS [5]	16 bytes	16 bytes	4	2 (16)	16	12

if no such keys exist. Recall that the keys are 128 bits long and that enumerating all of them to count those resulting in the targeted number of transactions is not feasible. To quantify memory leakage we need to generate symbolic constraints capturing all keys that result in a maximum number of shared memory transaction, and then to count/estimate the number of keys satisfying the constraints.

Generating constraints and choosing plaintexts. We did experiment with two GPU implementations (see Table 3). Each one of the two implementations performs 16 rounds in total. AES implementations strive to hide input values (plaintext and secret keys) by repeatedly mixing their values with those of the tables. Our tool could prune duplicate lines according to Section 4.4. We could build symbolic constraints for the maximum cliques for the first two rounds of each implementation. The generated constraints became too large beyond these rounds. As seen in Table 3, the constraints correspond to 12 unique lines for each implementation and represents the first 24 lines of the OpenSSL [6] implementation and the first 16 lines of the ISPASS [5] implementation. The size of the secret key is fixed to 16 bytes for AES-128. We decided to adopt 16 bytes plaintexts as they are the smallest size that the considered AES implementations handle. Each 16 bytes of the plain text are handled by 4 threads in the implementations. We used our conflict-guided approach to check the existence of a plaintext and a key corresponding to maximum numbers of shared memory transactions

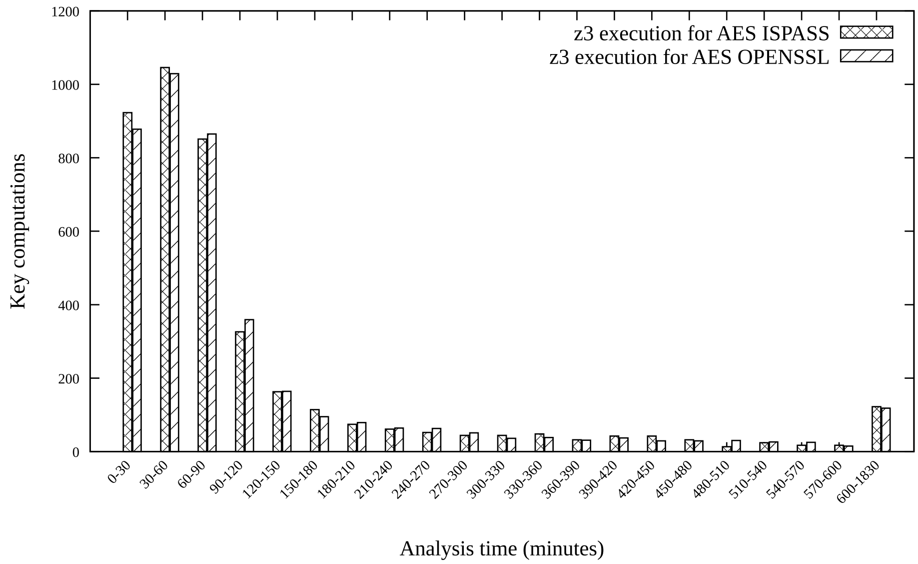


Fig. 6. Histogram of analysis times for AES ISPASS and OpenSSL versions. The X axis contains the interval for the analysis time in minutes. The Y axis shows how many of the analysis executions are in each interval.

during the first two rounds. This maximum number corresponds to the sum, over the considered lines of each implementation, of the maximum numbers of transactions possible for each line. For each implementation, we obtained, in less than 300 s, a concrete 16 bytes plaintext and a concrete 16 bytes key. We checked that they did indeed result in the maximum number of memory transactions. Now that we know this maximum number of transactions possible for the obtained plaintexts, we focus on quantifying the number of possible keys which lead to that maximum.

Counting and approximation. For each implementation, we use the plaintext found earlier and focus on quantifying the number of keys giving a maximum number of shared memory transactions. Recall the keys are 16 bytes long. Maintaining these 16 bytes as symbolic values in the generated constraints would characterize all keys resulting in a maximum number of transactions. A simple approach would be to query an SMT solver for concrete values of the 16 bytes that ensure the constraint is satisfied, to exclude these values in the next constraint, and to repeat until the constraint is unsatisfiable. This naive approach timed out in 48 h without finding a single key. We propose in the following to instead approximate the counting problem using multiple “smaller” SMT queries.

The idea is to generate several constraints instead of one, to check their satisfiability, and to deduce from the outcomes an upper bound on the number of possible keys. Intuitively, these constraints characterize subsets of the possible keys. Concretely, each constraint is obtained by setting one of the bytes $\mathtt{b} : 0 \leq \mathtt{b} < 16$ of the secret key to a concrete value $\mathtt{v} : 0 \leq \mathtt{v} < 256$ while maintaining the other 15 bytes symbolic. This results in $16 \times 256 = 4096$ constraints. Each with 15 symbolic bytes instead of the original 16. Intuitively, a constraint corresponding to setting byte `b` to value `v` characterizes all keys with `b` equal to `v` and resulting in a maximum number of shared memory transactions. If a corresponding query is unsatisfiable, then none of the $2^{15\times 8} = 2^{120}$ keys with the same fixed byte value can result in the targeted maximum number of shared memory transactions. An attacker observing this number of transactions can exclude all of the 2^{120} considered keys. We can launch the 4096 queries and set a timeout for each query. For each byte `b` this results in a partitioning of the 256 queries into $sat(\mathtt{b})$, $unsat(\mathtt{b})$ and $timed\_out(\mathtt{b})$ numbers of queries with $sat(\mathtt{b}) + unsat(\mathtt{b}) + timed\_out(\mathtt{b}) = 256$. An upper bound on the number of secret keys yielding the observation is then $\prod_{\mathtt{b}=0}^{255}(256 - unsat(\mathtt{b}))$.

Obtained results. We launched the 4096 queries in parallel on the Tetralith supercomputer [40] with hundreds of nodes each running CentOS 7 with two *Intel Xeon Gold 6130* and at least 96 GiB of RAM. We installed the SMT solver Z3 [41] version 4.4.1 on the nodes and exported the 4096 SMT queries from the internal KQuery representation of GKLEE to the standard SMT2 [42] format. We had one file per SMT query and it took 3 h to sequentially generate all SMT2 files. Individual files were approximately 50 MiB each for ISPASS and 30 MiB each for the OpenSSL version. We used a timeout of 32 h per query. As depicted by the histograms in Fig. 6, more than 75% of the queries terminated within two hours and 96% of them terminated within ten hours. Only nine (respectively 35) out of the 4096 queries timed out for the ISPASS implementation (respectively the OpenSSL implementation).

We report on the outcome of the SMT queries in Fig. 7. The figure shows, for each of the 16 bytes between 0 and 15, the number of queries (between 0 and 256) for which the SMT solver returned *UNSAT*. As can be observed from the figure, most queries were unsatisfiable. In fact, for each version, only one query per byte was satisfiable. The other queries being unsatisfiable or having timed out. That means that there are only a few combinations of keys that can lead to the maximum possible number of shared memory transactions for the chosen plaintexts. We obtained different key and plaintext values for the two versions. We also checked that the obtained concrete keys (when the SMT solver returned *SAT*) did indeed result in the maximum numbers of shared memory transactions for the corresponding implementation. Based on the number of *UNSAT* queries for each byte, we can deduce the following bounds:

- For the ISPASS implementation: there are at most $(256-245)^2 \times (256-248) \times (256-252)^2 \times (256-254)^2 \times (256-255)^9 < 2^{15.92}$ keys that may result in a maximum number of shared memory transactions. This represents an established leakage of at least 112.08 out of the 128 bits of secret information.
- For the OpenSSL implementation: there are at most $(256-253)^3 \times (256-254)^3 \times (256-255)^{10} < 2^{6.17}$ keys that may result in a maximum number of shared memory transactions.
  This represents an established leakage of at least 121.83 out of the 128 bits of secret information.

These results indeed confirm that very few keys may result in a maximal number of shared memory transactions. Recall that we conjectured, based on testing 30 000 randomly chosen keys as depicted in

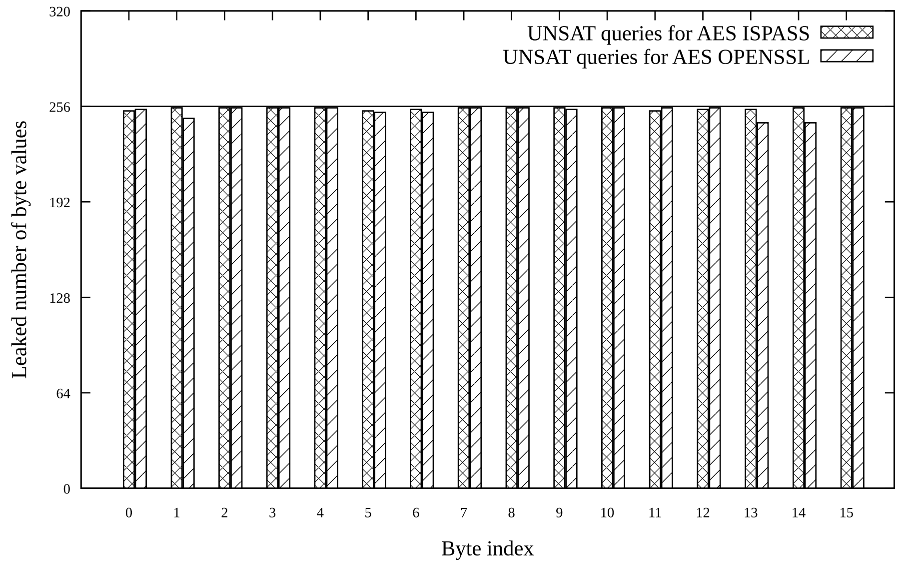


Fig. 7. Histogram of leaked values per byte for AES ISPASS and OpenSSL versions. The X axis specifies which byte index was set during the analysis. The Y axis shows how many of the analysis executions, out of 256 possible values, have returned unsatisfiable (UNSAT) from the Z3 solver analysis.

Fig. 5, that observing a maximal number of shared memory transactions would leak an important amount of information about the secret keys. Testing all possible 2^{128} keys to count how many result in the same maximal number of shared memory transactions is not feasible. Our on-demand conflict-guided approach allowed us to formally establish lower bounds (at least 112 bits out of the possible 128 bits) on the amount of information leakage.

7. Conclusion

We have introduced two encodings to symbolically reason about possible occurrences of shared memory bank conflicts in GPU programs. One encoding captures bank conflicts by focusing on the number of accesses to each shared memory word (used in the *direct* approach). The other encoding captures bank conflicts by focusing on the possible conflicts among threads of the same warp (used in the *conflict-guided* approach). We have proposed a refined version of the *conflict-guided* approach that proved to produce faster analysis times than the other approaches. We call this refined version the *on-demand conflict-guided* approach.

We have shown how our *on-demand conflict-guided* approach makes it possible to provide concrete program inputs that lead to different numbers of shared memory transactions. We give examples of how this can be used for both performance and security analysis of GPU programs. For performance analysis, our approach could provide inputs that directly impact execution time. For security analysis, our approach can be used to provide bounds on the amount of information leakage when a number of shared memory transactions is observed.

In future work we plan to further refine our *on-demand conflict-guided* approach with speed-ups regarding preprocessing of lines, memory management and multi-threaded GKLEE analysis.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research was partially supported by the Swedish Research Council under grant number 2017-04194. This research was also partially supported by the Singapore Ministry of Education (MOE) grant number MOE2018-T2-1-098 and National Research Foundation (NRF) grant number NRF2019-NRF-ANR092.

The computations and data handling were enabled by resources provided by the Swedish National Infrastructure for Computing (SNIC), partially funded by the Swedish Research Council through grant agreement no. 2018-05973.

References

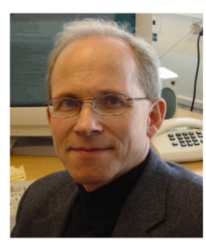
[1] J. Gomez-Luna, J.M. Gonzalez-Linares, J.I.B. Benitez, N.G. Mata, Performance modeling of atomic additions on GPU scratchpad memory, IEEE Trans. Parallel Distrib. Syst. 24 (11) (2012) 2273–2282.
[2] X. Mei, K. Zhao, C. Liu, X. Chu, Benchmarking the memory hierarchy of modern GPUs, in: IFIP International Conference on Network and Parallel Computing, Springer, 2014, pp. 144–156.
[3] X. Zhang, G. Tan, S. Xue, J. Li, K. Zhou, M. Chen, Understanding the gpu microarchitecture to achieve bare-metal performance tuning, in: Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2017, pp. 31–43.
[4] G. Pagès, B. Wilbertz, GPGPUs in computational finance: Massive parallel computing for American style options, Concurr. Comput.: Pract. Exper. 24 (8) (2012) 837–848.
[5] S.A. Manavski, CUDA compatible GPU as an efficient hardware accelerator for AES cryptography, in: 2007 IEEE International Conference on Signal Processing and Communications, IEEE, 2007, pp. 65–68.
[6] J. Gilger, J. Barnickel, U. Meyer, GPU-acceleration of block ciphers in the OpenSSL cryptographic library, in: D. Gollmann, F.C. Freiling (Eds.), Information Security, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 338–353.
[7] P. Harish, P.J. Narayanan, Accelerating large graph algorithms on the GPU using CUDA, in: International Conference on High-Performance Computing, Springer, 2007, pp. 197–208.
[8] S. Hong, S.K. Kim, T. Oguntebi, K. Olukotun, Accelerating CUDA graph algorithms at maximum warp, ACM SIGPLAN Not. 46 (8) (2011) 267–276.
[9] X. Sierra-Canto, F. Madera-Ramirez, V. Uc-Cetina, Parallel training of a back-propagation neural network using CUDA, in: 2010 Ninth International Conference on Machine Learning and Applications, IEEE, 2010, pp. 307–312.
[10] H. Grahn, N. Lavesson, M.H. Lapajne, D. Slat, CudaRF: a CUDA-based implementation of random forests, in: 2011 9th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA), IEEE, 2011, pp. 95–101.
[11] N. Mittal, S. Kumar, et al., Machine learning computation on multiple GPU's using CUDA and message passing interface, in: 2019 2nd International Conference on Power Energy, Environment and Intelligent Control (PEEIC), IEEE, 2019, pp. 18–22.
[12] CUDA toolkit documentation, 2021, URL http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.
[13] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, S.P. Rajan, GKLEE: concolic verification and test generation for GPUs, in: PPOPP, 2012, pp. 215–224.
[14] CUDA profiler page, 2021, URL https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[15] Radeon GPU profiler page, 2021, URL https://gpuopen.com/rgp/.
[16] Y. Iwase, D. Abe, T. Yakoh, GPGPU aided method for real-time systems, in: IEEE 10th International Conference on Industrial Informatics, IEEE, 2012, pp. 841–845.
[17] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, R. Rajkumar, RGEM: A responsive GPGPU execution model for runtime engines, in: 2011 IEEE 32nd Real-Time Systems Symposium, IEEE, 2011, pp. 57–66.
[18] A. Majumdar, L. Piga, I. Paul, J.L. Greathouse, W. Huang, D.H. Albonesi, Dynamic GPGPU power management using adaptive model predictive control, in: 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), IEEE, 2017, pp. 613–624.
[19] S. Hong, H. Kim, An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness, in: Proceedings of the 36th Annual International Symposium on Computer Architecture, 2009, pp. 152–163.
[20] S.S. Baghsorkhi, M. Delahaye, S.J. Patel, W.D. Gropp, W.-m.W. Hwu, An adaptive performance modeling tool for GPU architectures, in: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2010, pp. 105–114.
[21] I. Baldini, S.J. Fink, E. Altman, Predicting GPU performance from CPU runs using machine learning, in: 2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing, IEEE, 2014, pp. 254–261.
[22] M. Amaris, R.Y. de Camargo, M. Dyab, A. Goldman, D. Trystram, A comparison of GPU execution time prediction using machine learning and analytical modeling, in: 2016 IEEE 15th International Symposium on Network Computing and Applications (NCA), IEEE, 2016, pp. 326–333.
[23] T.T. Dao, J. Kim, S. Seo, B. Egger, J. Lee, A performance model for GPUs with caches, IEEE Trans. Parallel Distrib. Syst. 26 (7) (2014) 1800–1813.
[24] K. Hamaya, S. Yamane, et al., Detecting bank conflict of GPU programs using symbolic execution—Case study, J. Softw. Eng. Appl. 10 (02) (2017) 159.
[25] A. Betts, N. Chong, A.F. Donaldson, J. Ketema, S. Qadeer, P. Thomson, J. Wickerson, The design and implementation of a verification technique for GPU kernels, ACM Trans. Program. Lang. Syst. 37 (3) (2015) http://dx.doi.org/10.1145/2743017.
[26] P. Pereira, H. Albuquerque, I. da Silva, H. Marques, F. Monteiro, R. Ferreira, L. Cordeiro, SMT-based context-bounded model checking for CUDA programs, Concurr. Comput.: Pract. Exper. 29 (22) (2017) e3934.
[27] S. Blom, M. Huisman, M. Mihelčić, Specification and verification of GPGPU programs, Sci. Comput. Program. 95 (2014) 376–388.
[28] R. Alur, J. Devietti, O.S.N. Leija, N. Singhania, Static detection of uncoalesced accesses in GPU programs, Form. Methods Syst. Des. (2021) 1–32.
[29] Z.H. Jiang, Y. Fei, D. Kaeli, A novel side-channel timing attack on GPUs, in: Proceedings of the on Great Lakes Symposium on VLSI 2017, 2017, pp. 167–172.
[30] Z.H. Jiang, Y. Fei, D. Kaeli, Exploiting bank conflict-based side-channel timing leakage of GPUs, ACM Trans. Archit. Code Optim. (TACO) 16 (4) (2019) 1–24.
[31] B. Köpf, L. Mauborgne, M. Ochoa, Automatic quantification of cache side-channels, in: International Conference on Computer Aided Verification, Springer, 2012, pp. 564–580.
[32] G. Doychev, B. Köpf, L. Mauborgne, J. Reineke, Cacheaudit: A tool for the static analysis of cache side channels, ACM Trans. Inf. Syst. Secur. 18 (1) (2015) 4.
[33] C.S. Pasareanu, Q.-S. Phan, P. Malacaria, Multi-run side-channel analysis using symbolic execution and max-SMT, in: 2016 IEEE 29th Computer Security Foundations Symposium (CSF), IEEE, 2016, pp. 387–400.
[34] S. Chattopadhyay, M. Beck, A. Rezine, A. Zeller, Quantifying the information leakage in cache attacks via symbolic execution, ACM Trans. Embedded Comput. Syst. (TECS) 18 (1) (2019) 1–27.
[35] A. Horga, S. Chattopadhyay, P. Eles, Z. Peng, Genetic algorithm based estimation of non–functional properties for GPGPU programs, J. Syst. Archit. 103 (2020) 101697.
[36] L. De Moura, N. Bjørner, Satisfiability modulo theories: Introduction and applications, Commun. ACM 54 (9) (2011) 69–77.
[37] C. Barrett, A. Stump, C. Tinelli, The SMT-LIB Standard: Version 2.0, in: A. Gupta and D. Kroening (Eds.), Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK), 2010.
[38] CUDA toolkit samples, 2021, URL https://docs.nvidia.com/cuda/cuda-samples/index.html.
[39] E. Tromer, D.A. Osvik, A. Shamir, Efficient cache attacks on AES, and countermeasures, J. Cryptol. 23 (1) (2010) 37–71.
[40] Tetralith supercomputer page, 2021, URL https://www.nsc.liu.se/systems/tetralith/.
[41] Z3 SMT solver page, 2021, URL https://github.com/Z3Prover/z3.
[42] SMT2 language manual, 2021, URL https://smtlib.cs.uiowa.edu/language.shtml.

Adrian Horga is currently a Ph.D. student at the Department of Computer and Information Science (IDA), Linköping University. He received his M.Sc. in Information Technology at Politehnica University of Timişoara (UPT), Romania in 2013. His research interests include parallel computing, embedded systems, software analysis and testing.

Ahmed Rezine received his engineering degree from the Polytechnic School of Tunisia in 2002, and his Ph.D. degree in computer science from Uppsala University, Sweden in 2008. He was a Post-Doctoral Research Fellow both at the University of Paris Diderot, France and at Uppsala University. He is currently an Associate Professor of Computer Science at the Department of Computer and Information Science, Linköping University, Sweden. His current research interest includes extending the applicability of automated formal verification to larger and more complex computer systems. Dr. Rezine was a recipient of the Best Paper Award at the International Conference on Tools and Algorithms for the Construction and Analysis of Systems.

Sudipta Chattopadhyay is an Assistant Professor at the Information Systems Technology and Design (ISTD) Pillar at Singapore University of Technology and Design (SUTD). He received his Ph.D. in computer science from National University of Singapore (NUS) in 2013. His research interests include software analysis and testing, with a specific focus on designing efficient and secure software systems.



Petru Eles is Professor of Embedded Computer Systems with the Department of Computer and Information Science (IDA), Linköping University. Petru Eles' current research interests include embedded systems, real-time systems, electronic design automation, cyber–physical systems, hardware/software codesign, low power system design, fault-tolerant systems, design for test. He has published a large number of technical papers in these areas and co-authored several books. Petru Eles received several best paper awards at major Conferences.

Zebo Peng is currently Professor of Computer Systems, Director of the Embedded Systems Laboratory, and Vice-Chairman of the Department of Computer Science at Linköping University. His current research interests include design and test of embedded systems, electronic design automation, SoC testing, fault tolerant design, hardware/-software co-design, and real-time systems. He has published more than 350 technical papers and five books in these areas, and received four best paper awards and one best presentation award in major international conferences.