Noname manuscript No.
(will be inserted by the editor)

Automated Verification of Functional Correctness of Race-Free GPU Programs

Kensuke Kojima · Akifumi Imanishi · Atsushi Igarashi

Received: date / Accepted: date

Abstract We study an automated verification method for functional correctness of parallel programs running on graphics processing units (GPUs). Our method is based on Kojima and Igarashi's Hoare logic for GPU programs. Our algorithm generates verification conditions (VCs) from a program annotated by specifications and loop invariants, and pass them to off-the-shelf SMT solvers. It is often impossible, however, to solve naively generated VCs in reasonable time. A main difficulty stems from quantifiers over threads due to the parallel nature of GPU programs. To overcome this difficulty, we additionally apply several transformations to simplify VCs before calling SMT solvers.

Our implementation successfully verifies correctness of several GPU programs, including matrix multiplication optimized by using shared memory. In contrast to many existing verification tools for GPU programs, our verifier succeeds in verifying fully parameterized programs: parameters such as the number of threads and the sizes of matrices are all symbolic. We empirically confirm that our simplification heuristics is highly effective for improving efficiency of the verification procedure.

Keywords Program Verification · GPGPU · SMT · Symbolic Execution

1 Introduction

General-purpose computation on graphics processing units (GPGPU) is a technique to utilize GPUs, which consist of many cores running in parallel, to accelerate applications not necessarily related to graphics processing. GPGPU is one of the important techniques in high-performance computing, and has a wide range of applications [26]. However, it is hard and error-prone to hand-tune GPU

K. Kojima · A. Imanishi · A. Igarashi
Graduate School of Informatics, Kyoto University, Kyoto 606-8501, JAPAN

K. Kojima
Tel.: +81-75-753-5968
Fax: +81-75-753-4954
E-mail: kozima@fos.kuis.kyoto-u.ac.jp

programs for efficiency because the programmer has to consider cache, memory latency, memory access pattern, and data synchronization.

In this article, we study an automated deductive verification technique for functional correctness of GPU programs. We follow the standard method: our algorithm first generates verification conditions (VCs) from a program annotated with specification and loop invariants and then passes the generated VCs to off-the-shelf SMT solvers to check their validity. We empirically show that our technique can be applied to actual GPU programs, such as matrix multiplication programs optimized by using shared memory. Because shared memory optimization is a technique that is widely used when writing GPU programs, we believe that it is an encouraging result that we could verify a typical example of such programs.

We separate the verification problem into two parts: verifying race-freedom of general programs, and verifying functional correctness of race-free programs. We focus on the latter, relying on race detection techniques that have been studied elsewhere [21,2]. Race-freedom allows us to assume an arbitrary scheduling of threads without changing the behavior of a program. In particular, we can safely assume that all threads are executed in *complete lockstep* (that is, all threads execute the same instruction at the same time). Kojima and Igarashi [13,14] observed that such an assumption makes it possible to analyze GPU programs in a manner similar to sequential ones and developed Hoare logic for GPGPU programs executed in lockstep. We adapt their logic for VC generation.

Even under the race-freedom assumption, however, the generated VCs are often too complex for SMT solvers to solve in a reasonable amount of time. VCs tend to involve many quantifiers over threads and nonlinear expressions. Quantifiers over threads arise from assignment statements. When an assignment is executed on a GPU, it modifies more than one element of an array at a time. This means that the VC corresponding to an assignment should read "if there exists a thread writing into this index, ..., and otherwise, ..." In addition, the termination condition of a loop ("there is no thread satisfying the guard") also involves a quantifier over threads. Nonlinear expressions often appears in GPGPU programs as computation of offsets of arrays in a complicated way. This makes the verification problem harder because nonlinear integer arithmetic is undecidable in general. To overcome this difficulty, we devise several transformations to simplify VCs. Some of the simplification methods are standard (e.g., quantifier elimination) but others are specific to the current problem.

We implement a verifier for (a subset of) CUDA C, conduct experiments, and show that our method successfully verifies realistic GPU programs. Specifically, the correctness of optimized matrix multiplication programs using shared memory is verified, without instantiating parameters such as sizes of matrices and thread blocks. We also empirically confirm that our simplification heuristics is indeed highly effective to improve the verification process.

Contributions. Our main contributions are: (1) a VC generation algorithm for (race-free) GPU programs; (2) several simplification procedures to help SMT solvers discharge VCs; (3) implementation of a verifier based on (1) and (2); and (4) experiments to show that our verification method can indeed be applied to realistic GPU programs. Our approach can successfully handle fully parameterized programs, that is, we do not need to fix parameters such as the number of threads

and sizes of arrays, unlike much of the existing work (for example, GPUVerify [2] requires the user to specify the number of threads).

This article extends the previous work of the authors [15] with another simplification method (Section 4.4) and additional experiments.

Organization. The rest of the article is organized as follows. Section 2 explains the execution model of GPU programs on which our verification method is based. Section 3 describes our VC generation algorithm. Section 4 introduces several methods to simplify generated VCs. Section 5 reports our implementation and experimental results. Section 6 discusses related work, and finally we summarize the article and discuss future directions in Section 7.

2 Execution model of GPU programs

Compute Unified Device Architecture (CUDA) is a development environment provided by NVIDIA [27] for GPGPU. It includes a programming language CUDA C, which is an extension of C for GPGPU. A CUDA C program consists of host code, which is executed on a CPU, and device code, which is executed on a GPU. Host code is mostly the same as usual C code, except that it can invoke a function defined in device code. Such a function is called a *kernel function* (or simply kernel). The device code is also similar to usual C code, but it includes several special constants and functions specific to GPU, such as thread identifiers and synchronization primitives. The kernel function is executed on GPUs by the specified number of threads in parallel. The number of threads is specified in host code and does not change during the execution of a kernel function. When all the threads finish the execution, the result becomes available to host code. In this article we focus on the verification of kernel functions invoked by host code (it is allowed in CUDA C to call a kernel function from another kernel function, but we do not consider such a function call).

As mentioned in Section 1, we assume each instruction is executed in complete lockstep by all threads during the execution of device code. When the control branches during the execution, both branches are executed sequentially with threads irrelevant to branches being disabled. After both branches are completed, all the threads are enabled again. We say a thread is *inactive* if it is disabled, and *active* otherwise. This execution model is simplified from the so-called SIMT execution model, an execution model of CUDA C [27], in which threads form hierarchically organized groups and only threads that belong to the smallest group (called warp) are executed in lockstep. However, for race-free programs, there are not significant differences (except barrier divergence, which is an error caused by threads executing barrier synchronization at different program points).

Let us consider the kernel given in Figure 1, which we call ArrayCopy, and use it as a running example. This program copies the contents of a shared array (pointed to by) `a` to another shared array (pointed to by) `b`, both of length `len`. `N` is the number of threads, and `tid` is a thread identifier, which ranges from 0 to $\texttt{N}-1$. The first two lines specify a precondition and a postcondition, and the two lines above the loop declare loop invariants used for verification of the specification. These specifications will be used later but we ignore them for the moment because they are not used during the execution.

```
/*@ requires len == m * N;
    ensures  \forall int j; 0 <= j < len ==> b[j] == a[j]; */
void ArrayCopy (int *a, int *b, int len) {
  int i = tid;
  /*@ loop invariant i == N * loop_count + tid;
      loop invariant \forall int j; 0 <= j < N * loop_count ==> b[j] == a[j]; */
  while (i < len) {
    b[i] = a[i];
    i = i + N;
  }
}
```

Fig. 1 Running Example: ArrayCopy

If `len` is 6 and `N` is 4, the execution takes place as follows.[1] The local variable `i` is initialized to `tid`, so its initial value equals t at thread t ($0 \leq t < 4$). In the first iteration of the loop body, the first four elements are copied from `a` to `b`, and the value of `i` at thread t becomes $t + 4$. Then, the guard `i < len` is satisfied by only threads 0 and 1; therefore, threads 2 and 3 become inactive and the loop body is iterated again. Because active threads are only 0 and 1, the fourth and fifth elements of `a` are copied, and the values of `i` at threads 0, 1, 2, and 3 becomes 8, 9, 6, and 7, respectively. Now, no threads satisfy the guard, so the loop is exited and the program terminates with the expected result.

There are two typical constructs that are not included in the example: conditional (if-then-else) statements and barrier synchronization. If a conditional statement is encountered, both branches are executed serially. During the execution of then branch, the threads in which the condition is true are enabled, and other threads are enabled during the execution of else branch. It is not specified which of the branches are executed first, but the order of execution does not affect the result if the program is race-free (in our presentation in Section 3, we assume then branch is executed first). A barrier synchronization has no special meaning in our simplified execution model, in which the complete lockstep execution is assumed. However, we can treat it as an assertion that all threads are enabled at that point. Synchronization failure can be detected by regarding barrier synchronization as this type of assertion.

3 Verification Condition Generation

In this section, we describe how VCs are generated from a program annotated with specifications. We present verification generation as symbolic execution [12] of the axiomatic semantics of SIMT programs by Kojima and Igarashi [13, 14]. We do not review the previous work here but believe that the description below is detailed and self-contained enough, with the concrete execution model described in the last section in mind.

[1] We choose these initial values to explain what happens when the control branches. These initial values do not satisfy the precondition on the first line, so the asserted invariant is not preserved during execution.

3.1 Specifications

In general, an input to a VC generation algorithm is a program with annotations. There are three kinds of annotations: *preconditions*, *postconditions*, and *loop invariants* (or simply *invariants*), as in Figure 1. Preconditions and postconditions are conditions that should be satisfied by the initial and final state of the program, respectively. Loop invariants are associated for every loop construct appearing in the program, and specify conditions that should always be satisfied at the beginning and end of each iteration. Loop invariants are used by symbolic execution algorithm to handle the loop constructs. These three kinds of annotations are given as logical formulas on program variables, but in addition they may refer to auxiliary variables, called specification variables, which do not appear in the program.

As an example, let us take a look at the specification of ArrayCopy in Figure 1. The first line declares a precondition that the length of arrays is a multiple of the number of threads. A variable `m`, whose declaration is omitted, is a specification variable. We also assume implicitly that `a` and `b` do not overlap, and have length (at least) `len`. The second line declares the postcondition asserting that the contents of `a` are indeed copied into `b`. The loop contains two declarations of loop invariants. In the invariant we allow a specification variable `loop_count`, which stands for how many times the loop body has been executed. This variable is not present in CUDA C, but we have introduced it for convenience. It allows us to express the value of variables explicitly in an invariant. The first invariant specifies the value of the variable `i` on each iteration, and the second asserts that at the beginning of l-th iteration (counting from 0) the first $N \cdot l$ elements of `a` have been already copied to `b`.

3.2 Symbolic Execution

We use a technique called symbolic execution to generate a VC. Symbolic execution translates a program into a set of logical formulas Γ that encodes the execution of the program. In particular, Γ contains information about possible final states of the program. Therefore, in order to verify the correctness of the program, it is sufficient to check that the generated set Γ implies the postconditions. This implication condition is the output of the VC generation algorithm. Several additional formulas are also generated (and such formulas are also called VCs) during the symbolic execution. They are to ensure that the provided loop invariants are indeed true at the beginning and the end of each iteration. In this manner, the verification problem is systematically translated to the validity checking problem, which can be passed to off-the-shelf SMT solvers.

3.3 Details of the VC Generation Algorithm

We explain the details of our VC generation algorithm using the example ArrayCopy in Figure 1. Constructs that do not appear in this example are explained at the end of the section.

First, generate specification variables i_0 and len_0, which represent the initial values of `i` and `len`, respectively, and a_0 and b_0, which represent the contents of

arrays pointed to by `a` and `b`, respectively. Here, i_0, a_0, and b_0 has the type of maps from `int` to `int`, and len_0 has type `int`. Since a_0 and b_0 represent arrays, they are naturally represented as maps. The reason that i_0 also has a map type is that it corresponds to a local variable whose value varies among threads. So, expression $i_0(t)$ stands for the value of `i` at thread t. We also need m which is a specification variable of type `int`. The precondition in the first line is translated into the formula $len_0 = m \cdot N$, so we assume this equation holds. In the next line the value of `i` is updated to `tid` in all threads. In general every time we encounter an assignment we introduce a new variable that represents the value of the variable being assigned after this assignment. In the case of `i = tid` we introduce a new variable i_1 of the same type as i_0, and assume $\forall t.0 \leq t < N \to i_1(t) = t$, that is, its value on thread t equals t. For later use, let us denote by Γ_{entry} the list consisting of the two constraints we have introduced so far:

$$\Gamma_{\text{entry}} \stackrel{\text{def}}{=} len_0 = m \cdot N, \forall t.0 \leq t < N \to i_1(t) = t.$$

So, Γ_{entry} represents possible states of the program at the beginning of the loop. Since two invariants are declared in this loop, we have to check that they are true at the entry, so we generate two conditions to be verified:

$$\Gamma_{\text{entry}} \vdash \forall t.0 \leq t < N \to i_1(t) = N \cdot 0 + t, \tag{T1}$$

$$\Gamma_{\text{entry}} \vdash \forall j.0 \leq j < N \cdot 0 \to b_0(j) = a_0(j). \tag{T2}$$

Below we call a condition of the form $\Gamma \vdash \varphi$ a *task*, and φ the *goal*. Tasks (T1) and (T2) assert that the first and second invariants are true at the loop entry, respectively. The right-hand sides of these tasks are obtained from loop invariants by simply replacing `loop_count` with 0, the initial value of the loop counter.

Next, we have to encode the execution of the loop, but in general it is impossible to know how many times the loop body is executed. Rather than iterating the loop, we directly generate a constraint that abstracts the final state of the loop, relying on the invariants supplied by the programmer [10]. Also we have to verify that the supplied invariants are indeed preserved by iterating the loop. To do this we first introduce a new variable for each program variable being modified in the loop body. In the case of our example, variables being modified are `b` and `i`, so we generate fresh b_1 and i_2. We also introduce l corresponding to the loop counter. Let Γ_{loop} be the following list of formulas:

$$\begin{aligned}\Gamma_{\text{loop}} \stackrel{\text{def}}{=} \Gamma_{\text{entry}}, 0 \leq l, \forall t.0 \leq t < N \to i_2(t) = N \cdot l + t,\\ \forall j.0 \leq j < N \cdot l \to b_1(j) = a_0(j).\end{aligned}$$

Γ_{loop} consists of three additional constraints. The first one, $0 \leq l$, says that the loop counter is not negative. The second and third ones correspond to invariants, and they assert that invariants are true for variables b_1, i_2, and l we just have introduced. Note that in Γ_{loop} it is not yet specified whether the loop is already exited or not.

Consider the case the loop is continued. Then, there is at least one thread that satisfies the loop guard `i < len`, which is expressed: $\exists t.0 \leq t < N \wedge i_2(t) < len_0$. Since the loop body contains assignments to `b` and `i`, we generate new variables b_2 and i_3 and add constraints expressing that these variables are the result of executing these assignments. Writing down such constraints is a little more involved than

before, because these assignments are inside the loop body, and therefore there may be several threads that are inactive (actually in this example such a situation never happens, but to describe how VCs are generated in a general case, let us proceed as if we do not know this fact). We use the notation $\mathit{assign}(b_2, i_2 < \mathit{len}_0, b_1, i_2, a_0(i_2))$ for such a constraint.[2] This intuitively means that b_2 is the result of executing `b[i] = a[i]` with the values of `b` and `i` being b_1 and i_2 respectively, and active threads t being precisely those that satisfy $i_2(t) < \mathit{len}_0$. The first argument is the new value of the variable being assigned, the second specifies which threads are active, the third is the original value of the variable being assigned, the fourth is the index being written (in general, this is an n-tuple if the array being assigned is n-dimensional, and the 0-tuple $\cdot$ if the variable is scalar), and the last is the value of the right-hand side of the assignment. It can be written out as

$$\begin{aligned}\forall n.\, (\exists t.0 \le t < N \land i_2(t) < \mathit{len}_0 \land i_2(t) = n \land b_2(n) = a_0(i_2(t))) \lor \\ ((\forall t.\neg(0 \le t < N \land i_2(t) < \mathit{len}_0 \land i_2(t) = n)) \land b_2(n) = b_1(n))\,,\end{aligned} \tag{1}$$

but the concrete definition does not matter here. For general cases, readers are referred to Kojima and Igarashi [13,14]. Putting these constraints together we obtain Γ_{iter} defined as follows:

$$\begin{aligned}\Gamma_{\mathrm{iter}} \stackrel{\mathrm{def}}{=} \Gamma_{\mathrm{loop}}, \exists t.0 \le t < N \land i_2(t) < \mathit{len}_0, \\ \mathit{assign}(b_2, i_2 < \mathit{len}_0, b_1, i_2, a_0(i_2)), \mathit{assign}(i_3, i_2 < \mathit{len}_0, i_2, \cdot, i_2 + N).\end{aligned}$$

Using Γ_{iter} we can write the tasks corresponding to the invariant preservation as follows:

$$\Gamma_{\mathrm{iter}} \vdash \forall t.0 \le t < N \to i_3(t) = N \cdot (l+1) + t, \tag{T3}$$

$$\Gamma_{\mathrm{iter}} \vdash \forall j.0 \le j < N \cdot (l+1) \to b_2(j) = a_0(j). \tag{T4}$$

The right-hand sides of these tasks are obtained by replacing `loop_count`, `b`, and `i` in the invariants with their values after the iteration, namely $l+1$, b_2, and i_3, respectively.

Finally we consider the case loop is exited, in which case the loop guard is false in all threads. Therefore we put

$$\Gamma_{\mathrm{exit}} \stackrel{\mathrm{def}}{=} \Gamma_{\mathrm{loop}}, \forall t.0 \le t < N \to \neg(i_2(t) < \mathit{len}_0).$$

Since there are no more statements to be executed, it only remains to verify that the postcondition holds under this constraint. So the final task is as follows:

$$\Gamma_{\mathrm{exit}} \vdash \forall j.0 \le j < \mathit{len}_0 \to b_1(j) = a_0(j). \tag{T5}$$

To summarize, we generate tasks (T1–T5) as VCs for our example program. (T1) and (T2) ensure that the invariants hold when the loop is entered, (T3) and (T4) ensure that the invariants are preserved by executing the loop body, and (T5) ensures that the postcondition is satisfied when the program terminates.

Finally let us mention two more constructs: conditional statements and barrier synchronization. As mentioned before, a conditional statement is executed

[2] Some of the terms appearing in this expression are not well-typed. We could write $\mathit{assign}(b_2, (\lambda t.i_2(t) < \mathit{len}_0), b_1, (\lambda t.i_2(t)), (\lambda t.a_0(i_2(t))))$, but for brevity we abbreviate it as above.

sequentially with switching active threads. When a statement `if` b `then` P `else` Q is encountered, we first process P, and then Q (because we assume race-freedom, the order does not matter). When processing P we have to bear in mind that active threads are restricted to those at which b evaluates to true, and similarly for Q. Barrier synchronization is, since we assume the execution is complete lockstep, considered as an assertion that all threads are active at that program point. We can generate an extra task $\Gamma \vdash \forall t.0 \leq t < N \to \mu(t)$, where $\mu(t)$ is a formula expressing that thread t is currently active, to verify that the synchronization does not fail. For example, if there were synchronization at the end of the loop body in ArrayCopy, $\mu(t)$ would be $i_2(t) < len_0$.

4 Simplifying Verification Conditions

Unfortunately, SMT solvers often fail to discharge VCs generated by the algorithm described in the previous section. In this section, we describe several schemes to simplify VCs used in our verifier implementation. The rewriting schemes introduced in this section are, except for the ones explicitly mentioned, sound and complete in the sense that the VC obtained by applying them is valid if and only if the original VC is valid. This is because they always replace a formula with equivalent one.

The main difficulty stems from universal quantifiers, which are typically introduced by assignment statements and loop invariants. When these universally quantified formulas are put on the left-hand side of the tasks, the solvers have to instantiate them with appropriate terms, but it is often difficult to find them. To overcome this difficulty, in Sections 4.1 and 4.2 we introduce two strategies, which we call ElimAssign and Rewrite, that find appropriate instances of these quantified variables and rewrite VCs using these instances.

Another difficulty stems from multiplication over integers that often arises from indices of arrays. This makes VCs harder to discharge automatically, since nonlinear integer arithmetic is undecidable (even without quantifiers). The transformations MergeQuant and SimpAffine, described in Section 4.3 and 4.4, respectively, simplify formulas involving quantifiers and/or nonlinear formulas in a certain form.

A standard approach to the first problem would be to annotate a quantifier with *triggers* [8, Section 5.1], which is an expression involving quantified variables, to give SMT solvers hints. Triggers are used to decide to which expression those variables are instantiated. For example, if a trigger $f(x)$ is associated to a quantified formula $\forall x.\varphi(x)$ which is a antecedents of the current task, and $f(t)$ appears somewhere in the task, then the variable x is instantiated with t, and thus $\varphi(t)$ is added as a new antecedents of the task. It is natural to conjecture that we can help SMT solvers by providing triggers.

However, as far as we have tried, it is not sufficient to simply provide triggers to quantifiers. This is because MergeQuant and SimpAffine, which reduce the degree of nonlinearity, often works only after ElimAssign and Rewrite are applied, and hence they have to be performed before the other two transformations. Because all of our transformations are performed before calling off-the-shelf SMT solvers, it is not straightforward to replace them with triggers.

4.1 Eliminating *assign*

We first introduce a transformation which we call ElimAssign. During VC generation, we introduce a new assumption involving *assign* for each assignment statement. As we have seen in (1), *assign* is universally quantified and therefore has to be instantiated by appropriate terms. The main objective of ElimAssign is to find all necessary instances automatically, and rewrite the VC using such instances (as a result, *assign* may be removed from the task). Since (1) is introduced to specify the value of b_2, we instantiate (1) by every term u such that $b_2(u)$ appears in VCs. By enumerating such u's (including those inside quantifiers) we would find all instances for n that are necessary to prove VCs.

There are two cases to consider: assignments to local variables and shared variables. As an example of the local case, let us consider i_3 appearing in (T3). Its value is specified by $\mathit{assign}(i_3, i_2 < \mathit{len}_0, i_2, \cdot, i_2 + N)$ in Γ_{iter}, which implies: (a) if t is a thread ID that is active (that is, $i_2(t) < \mathit{len}_0$), then the value of i_3 at t is $i_2(t) + N$, and (b) otherwise the value of i_3 at t is $i_2(t)$. In case (a), $i_3(t) = N \cdot (l+1) + t$ is equivalent to $i_2(t) + N = N \cdot (l+1) + t$, and in case (b) it is equivalent to $i_2(t) = N \cdot (l+1) + t$. Therefore by doing case splitting, we can rewrite the right-hand side of (T3) into:

$$\begin{aligned}&\forall t.(0 \le t < N \to i_2(t) < \mathit{len}_0 \to i_2(t) + N = N \cdot (l+1) + t) \wedge \\ &\quad (0 \le t < N \to \neg(i_2(t) < \mathit{len}_0) \to i_2(t) = N \cdot (l+1) + t).\end{aligned}$$

The first and the second conjuncts correspond to cases (a) and (b), respectively.

For the case of shared variables, consider b_2 in task (T4). Similarly to the previous case, for each j either (a) there exists a thread t such that $i_2(t) < \mathit{len}_0$, $i_2(t) = j$, and $b_2(j) = a_0(i_2(t))$, or (b) there is no such thread t, and $b_2(j) = b_1(j)$. We obtain the following formula by rewriting the right-hand side of (T4):

$$\begin{aligned}&\forall j.(0 \le j < N \cdot (l+1) \to \\ &\qquad \forall t.0 \le t < N \wedge i_2(t) < \mathit{len}_0 \wedge i_2(t) = j \to a_0(i_2(t)) = a_0(j)) \wedge \\ &\quad (0 \le j < N \cdot (l+1) \to \\ &\qquad (\forall t.\neg(0 \le t < N \wedge i_2(t) < \mathit{len}_0 \wedge i_2(t) = j)) \to b_1(j) = a_0(j)).\end{aligned} \tag{2}$$

Following this strategy we can rewrite the VC so that the first argument of *assign* does not appear in the resulting VC, thus SMT solvers do not have to search for instances of *assign* any more.

4.2 Applying Equalities in Antecedents

Invariants often involve a quantified and guarded equality that specifies the values of program variables, as we can see in ArrayCopy. We illustrate how to rewrite a formula using such an equality, and why such a rewriting is helpful in simplifying VCs. The method described below applies to both goals and assumptions.

Consider b_1 in the task (T5). Using the invariant $\forall j.0 \le j < N \cdot l \to b_1(j) = a_0(j)$, we can rewrite $b_1(j)$ into $a_0(j)$, but only under the assumption that $0 \le$

$j < N \cdot l$. Taking this condition into account, we can see that the goal $\forall j.0 \leq j < len_0 \to b_1(j) = a_0(j)$ can be changed to:

$$\begin{aligned}\forall j.0 \leq j < len_0 \to (0 \leq j < N \cdot l \wedge a_0(j) = a_0(j)) \vee \\ (\neg(0 \leq j < N \cdot l) \wedge b_1(j) = a_0(j)).\end{aligned} \tag{3}$$

After this transformation, we can use several simplifications to transform the task into an easier one that can be solved automatically. Let us demonstrate how this can be done. We have both $\forall t.0 \leq t < N \to \neg(i_2(t) < len_0)$ and $\forall t.0 \leq t < N \to i_2(t) = N \cdot l + t$ in Γ_{exit}; therefore, rewriting $i_2(t)$ in the same way as above, we can see that it follows from Γ_{exit} that

$$\begin{aligned}\forall t.0 \leq t < N \to \neg\big((0 \leq t < N \to N \cdot l + t < len_0) \wedge \\ (\neg(0 \leq t < N) \to i_2(t) < len_0)\big).\end{aligned}$$

By using laws of propositional logic we can simplify this as $\forall t.0 \leq t < N \to \neg(N \cdot l + t < len_0)$, and by eliminating the quantifier we obtain $len_0 \leq N \cdot l$. From this, (3) is easily derived by SMT solvers.

Similarly, (2) can be simplified as follows: the first conjunct is easily proved; in the second conjunct we can replace $i_2(t)$ with $N \cdot l + t$, and then eliminate $\forall t$ to obtain

$$\forall j.0 \leq j < N \cdot (l+1) \to \neg(0 \leq j - N \cdot l < N \wedge j < len_0) \to b_1(j) = a_0(j).$$

In general, we first search for an assumption of the form

$$\forall x_1.\gamma_1 \to \forall x_2.\gamma_2 \to \ldots \to \forall x_m.\gamma_m \to f(s_1, \ldots, s_n) = s' \tag{4}$$

where f is a function symbol. For each such assumption, find another formula (either one of the assumptions or the goal) in which f occurs. Such a formula can be written as $\psi[\varphi(f(t_1, \ldots, t_n))]$, where every variable occurrence of $t_1, \ldots, t_n$ is free in $\varphi(f(t_1, \ldots, t_n))$. Then by rewriting f we obtain:

$$\begin{aligned}\psi[\,&\big(\exists x_1 \ldots x_m.\gamma_1 \wedge \cdots \wedge \gamma_m \wedge s_1 = t_1 \wedge \cdots \wedge s_n = t_n \wedge \varphi(s')\big) \vee \\ &(\forall x_1 \ldots x_m.\neg\left(\gamma_1 \wedge \cdots \wedge \gamma_m \wedge s_1 = t_1 \wedge \cdots \wedge s_n = t_n\right)) \wedge \varphi(f(t_1, \ldots, t_n))].\end{aligned} \tag{5}$$

Intuitively, this can be read as follows. If there are $x_1, \ldots, x_n$ that satisfy $\gamma_1, \ldots, \gamma_n$ and $s_i = t_i$ for every i, then by (4) we can replace $\varphi(f(t_1, \ldots, t_n))$ with $\varphi(s')$ (the first disjunct). If there are no such $x_1, \ldots, x_n$, then we leave $\varphi(f(t_1, \ldots, t_n))$ unchanged (the second disjunct).

If this rewriting is unconditionally repeated until these is no expression that can be rewritten, then the procedure does not necessarily terminate. This is because f may appear in γ_i or φ in (5), aside from the explicitly written occurrence at the end of this formula. In practice, we could limit the number of repetition to avoid divergence, but we are not aware of an appropriate limit at the time of writing. We are aware of an example that indeed causes this situation, but none of the programs shown in Section 5 causes this situation during the experiments.

```
/*@ requires len == m * N;
    ensures  \forall int j; 0 <= j < len ==> b[j] == a[j]; */
void ArrayCopyMultBlocks (int *a, int *b, int len) {
  i = bid * bsize + tid;
  /*@ loop invariant i == N * loop_count + bid * bsize + tid;
      loop invariant
        \forall int j; 0 <= j < N * loop_count ==> b[j] == a[j]; */
  while (i < len) {
    b[i] = a[i];
    i = i + N;
  }
}
```

Fig. 2 A variant of the running example: ArrayCopyMultipleBlocks

4.3 Merging Quantifiers

Aside from standard transformations on formulas such as quantifier elimination, we introduce a procedure MergeQuant that replaces two quantifiers with a single one. Typical example is the following: if x and y range over integers, $\forall x.0 \leq x < a \to \forall y.0 \leq y < b \to \varphi(x + ay)$ (or equivalently, $\forall x.0 \leq x \leq a - 1 \to \forall y.0 \leq y \leq b - 1 \to \varphi(x + ay)$) is equivalent to $0 < a \to \forall z.0 \leq z < ab \to \varphi(z)$ (the antecedent $0 < a$ is necessary because otherwise if both a and b are negative the former is trivially true while the latter would not). This pattern often arises from indices of an array.

Let us illustrate how this helps simplify a VC. This transformation typically applies when a thread hierarchy and/or two-dimensional arrays are involved. Consider the program in Figure 2. Here we assume that threads are grouped into *blocks*, as in actual CUDA C or OpenCL programs. Each block consists of an equal number of threads. In the program above, `bsize` is the number of threads contained in one block, and `bid` is the identifier for a block, called block ID. When `bid` is evaluated on a certain thread, the result is the block ID of the block to which the thread belongs. `N` is, as before, the number of threads, and now equals the product of `bsize` and the number of blocks.

Let us consider the termination condition of the loop:

$$\forall t.0 \leq t < T \to \forall b.0 \leq b < B \to \neg(N \cdot l + b \cdot T + t < len)$$

where T denotes the number of threads per block, and B the number of blocks (we replaced `i` with $N \cdot l + b \cdot T + t$ using the first invariant). By merging two quantifiers, we obtain

$$0 < T \to \forall z.0 \leq z < T \cdot B \to \neg(N \cdot l + z < len).$$

The quantification over z is now easily eliminated, and we obtain $0 < T \to T \cdot B \leq 0 \vee len \leq N \cdot l$.

Up to now we have assumed that the quantifiers that can be merged have the form $\forall x.0 \leq x < a \to \ldots$, but in general this is not the case. Other simplification procedures (quantifier elimination, in our implementation) may convert formulas to their normal forms. After that, the guard $0 \leq x < a$ may be modified, split, or moved to other places. This significantly makes the general algorithm complicated. Because guards do not necessarily follow quantifiers, it is not straightforward to find a pair of quantifiers that can be merged as described above.

Our strategy in the general case is the following. (I) For every quantified subformula $\forall x.\varphi(x)$, find a such that $\forall x.\varphi(x)$ is equivalent to $\forall x.0 \le x < a \to \varphi(x)$. We call such a a *bound* of x. (II) For each subformula $\forall x.\forall y.\varphi(x,y)$, where x and y have bounds a and b, respectively, find $\psi(z)$ such that $\varphi(x,y)$ is equivalent to $\psi(x+ay)$ (or $\psi(y+bx)$). Then we can replace $\forall x.\forall y.\varphi(x,y)$ with an equivalent formula $0 < a \to \forall z.0 \le z < ab \to \psi(z)$, as desired. For the existential case, use $\wedge$ instead of $\to$. There may be multiple (actually infinitely many) bounds, and only some of them can be used as a in step (II). We collect as many bounds as possible in step (I), and try step (II) for every bound a of x we found. Below we simply write φ rather than $\varphi(x)$ if no confusion arises.

For step (I), note that if $\neg(0 \le x)$ implies φ and $\neg(x < a)$ implies φ, then $\forall x.\varphi$ if and only if $\forall x.0 \le x < a \to \varphi$. Similarly, if φ implies both $0 \le x$ and $x < a$, then $\exists x.\varphi$ if and only if $\exists x.0 \le x < a \wedge \varphi$. Therefore we can split the problem as follows: for the universal case, (i) check that $\neg(0 \le x)$ implies φ, and (ii) find a such that $\neg(x < a)$ implies φ; for the existential case, (i) check that φ implies $0 \le x$, and (ii) find a such that φ implies $x < a$. Because both of them can be solved similarly, we shall focus on (ii).

Let us say that a is a $\forall$-bound ($\exists$-bound) of x in φ if $\neg(x < a)$ implies φ (φ implies $x < a$, respectively). Then we are to find $\forall$- and $\exists$-bounds of x in a given φ. The procedure is given recursively. If φ is atomic, then the problem is easy, although there are tedious case distinctions. For example, $\forall$-bound of $x \ge t$ is t,[3] $\forall$-bound of $x < t$ does not exist, and $\exists$-bound of $x \le t$ is $t+1$. If φ is atomic but not an inequality, then we consider there are no bounds. If φ is $\varphi_1 \wedge \varphi_2$, then $\forall$-bounds of φ is the intersection of those of φ_1 and φ_2 (this may miss some bounds, but we confine ourselves to this approximation), and $\exists$-bounds are the union of those of φ_1 and φ_2. The $\forall$- and $\exists$-bounds of $\neg\varphi$ are $\exists$- and $\forall$-bounds of φ, respectively. Bounds of $\forall y.\varphi$ are those of φ. We omit $\vee$, $\to$, and $\exists$ since they are derived from other connectives by the laws of classical logic.

Step (II) is done by verifying that all atomic formulas depends only on $x+ay$. First, consider $s(x,y) < t(x,y)$ where s and t are polynomials in x, y. There is a simple sufficient condition: if there exists a polynomial $u(z)$ such that $t(x,y) - s(x,y) = u(x+ay)$, then $s(x,y) < t(x,y)$ is equivalent to $0 < u(x+ay)$. Therefore it is sufficient to check that $t(x,y) - s(x,y)$ can be written as a polynomial of $x+ay$, which is not difficult. Indeed, a polynomial $p(x,y)$ can be written in the form $u(x+ay)$ if and only if $p(x,y) = p(x+ay, 0)$, and in this case u is given by $u(z) = p(z,0)$ (apply this fact to the case $p(x,y) = t(x,y) - s(x,y)$). If s and t are not polynomials, or a predicate other than inequalities is used, then we check whether all arguments of the predicate or function symbols can be written as $u(x+ay)$.

4.4 Simplifying Affine Expressions

We devise another simplification strategy for nonlinear formulas, which we call SimpAffine. For example, if all variables range over integers,

$$0 \le x < a \wedge 0 \le x' < a \wedge x + ay = x' + ay' \to x = x' \tag{6}$$

[3] In this case $t+1, t+2, \ldots$ are also $\forall$-bounds, but we do not take them into account. Practically, considering only t seems sufficient in many cases.

is valid, but proving this formula is not easy for SMT solvers, as far as we have tried. SimpAffine transforms this formula into

$$0 \le x < a \land 0 \le x' < a \land x = x' \land y = y' \to x = x', \tag{7}$$

which is easily proved by SMT solvers. Similar patterns frequently arise from the computation of indices of arrays during the verification, and therefore a simplification method for this kind of formulas is useful.

SimpAffine first finds bounds of variables, similarly to step (I) of MergeQuant. Then, using the information of bounds, it tries to simplify each atomic formula $t \bowtie t'$, where $\bowtie$ is either $=$, $<$, $>$, $\le$, or $\ge$. Given an atomic formula $t \bowtie t'$, first it is checked whether t and t' have a nontrivial common factor. If they have a common factor u, then $t \bowtie t'$ can be simplified using

$$\begin{aligned} t = t' &\iff t/u = t'/u \lor u = 0, \\ t \le t' &\iff (t/u \le t'/u \land 0 \le u) \lor (t/u \ge t'/u \land 0 \ge u), \end{aligned}$$

and similar variants for other three predicates. If t and t' do not have a common factor, then the algorithm checks whether this formula has one of the following forms (up to equivalence):

1. $at + x \bowtie at' + x'$,
2. $at - x \bowtie at' - x'$, or
3. $at \pm x \bowtie at'$,

where a is a bound of both x and x'. The first form can be simplified using

$$\begin{aligned} at + x = at' + x' &\iff t = t' \land x = x', \\ at + x \le at' + x' &\iff t < t' \lor (t = t' \land x \le x'), \\ at + x < at' + x' &\iff t < t' \lor (t = t' \land x < x'), \end{aligned}$$

and similarly for $\ge$ and $>$. The formula (7) is obtained from (6) by applying the first relation. The second form can be reduced to the first form: $at + x' \bowtie at' + x$. The third one can be handled by substituting $x' = 0$ into the first case, and using $at - x \bowtie at' \iff at \bowtie at' + x$. However, in several cases we can write the result more explicitly as

$$\begin{aligned} at + x \le at' &\iff t < t' \lor (t = t' \land x = 0) \\ at + x < at' &\iff t < t', \\ at + x \ge at' &\iff t \ge t' \end{aligned}$$

which are simpler.

SimpAffine affects MergeQuant because, if a formula containing $ax+y$ is rewritten into a formula containing isolated occurrences of x and y, then the MergeQuant do not work for the resulting formula (since it requires these variables to occur only in the form $ax + y$). Similarly, MergeQuant affects SimpAffine, because if $\forall xy.(\dots ax + y \bowtie ax' + y' \dots)$ is modified as $\forall z.(\dots z \bowtie ax' + y' \dots)$, then SimpAffine does not apply to the second formula.

Therefore, the order the two simplifications are applied affects the result of verification. Because we need both of them in our experiments (details are discussed in Section 5), we use both of them and generate two separate tasks. The

original task is valid if at least one of the two tasks is proved (and in this case, both of them are in fact valid, although SMT solvers are not necessarily able to prove both of them).

4.5 Additional Heuristics

It is sometimes the case that the simplified goal is not still provable by SMT solvers, but the following transformations help proving the task (they are sound but not complete, i.e. they may replace a valid goal with an invalid one).

- If an equality $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$ occurs in a positive position, then we may replace it with $s_1 = t_1 \wedge \cdots \wedge s_n = t_n$.
- A subformula occurring in a positive (negative) position of a task may be replaced by False (True, respectively). We try this for a subformula of the form $f(t_1, \ldots, t_n) = t$ where f corresponds to a program variable.

It is likely that SMT solvers can perform these rewriting, but application of these schemes prior to calling SMT solvers increased the number of tasks that can be solved by SMT solvers, as far as we have tried. It appears that the simplification methods works better if these schemes are applied, and therefore including them in the implementation can improve performance.

5 Implementation and Experiment

5.1 Implementation

We have implemented the method described above and conducted an experiment on several kernels. Our implementation takes source code annotated with specifications (pre- and post-conditions and loop invariants) as an input and checks whether the specification is satisfied. The input language is a subset of CUDA C, but we slightly modified the syntax so that we can use an existing C parser without modification. This is just to simplify the implementation. A subset of ANSI/ISO C Specification Language (ACSL) [4] can be used to describe specifications.

The verifier first generates VCs as described in Section 3, and performs the simplification in Section 4 roughly in the following order: (1) ElimAssign (Section 4.1); (2) rewriting (Section 4.2); (3) MergeQuant (Section 4.3) and/or SimpAffine (Section 4.4). The last step returns two alternative tasks for each task, one of which is obtained by applying both of MergeQuant and SimpAffine in this order, and the other is obtained by applying only SimpAffine. If these two tasks are identical, to avoid the redundancy only one task is returned.

In addition to these operations, we also use standard simplification methods such as quantifier elimination. After simplifying the tasks, several SMT solvers are called on each task, and run in parallel. Each task is considered completed when one of the solvers successfully proves it. If there are tasks that none of the solvers can prove, then the heuristics in Section 4.5 are applied, and then SMT solvers are called again. This step is repeated at most 10 times. If there is still a task that remains unsolved after 10 repetitions, the verification fails.

[4] `https://frama-c.com/acsl.html`

The front-end is written in OCaml. We use Cil [25] to parse the input, and the syntax tree is converted into tasks using Why3 [4] API. Simplification of formulas is implemented as a transformation on data structures of Why3, and SMT solvers are called through Why3 API functions. We use Alt-Ergo, CVC3, CVC4, E Theorem Prover, and Z3 as back-ends.[5] Although Why3 provides a programming language WhyML, currently we use Why3 only for manipulating formulas and calling SMT solvers.

The source code of our tool called Vericuda and the benchmark programs used in our experiment is available via `https://github.com/SoftwareFoundationGroupAtKyotoU/Vericuda`.

5.2 Experiments

Using our implementation, we have verified the functional correctness of the following programs.

`scp` Copies an array contents to another array, but shifting by one element to the right (the last element of the input array is copied to the first position of the output array). A similar program appears as a running example in [3], and also provided as an example of VerCors Toolset.[6]

`rotate` Rotates an array in place by one element to the right. This is essentially the same as rotate in [1].

`vectorAdd` Receives three arrays, and outputs the sum of the first two (as vectors) into the third array. The algorithm is similar to ArrayCopy.

`matrixMul` Computes the multiplication of two matrices and output the result into another two-dimensional array. Each thread computes one element of the output matrix. This program uses a common optimization technique with shared memory. This is taken from NVIDIA CUDA Samples [27] and slightly modified without changing the essential part of the algorithm.

`matrixMul2` Similar to `matrixMul`, but each thread computes two elements of the output matrix, so that it improves `matrixMul` by hiding memory latency. The postcondition for this program is not written in the most natural form, because in our current implementation the postcondition is written in a form which is better handled by SMT solvers.

`matrixMul4` Similar to `matrixMul2`, but each thread processes four elements.

`diffusion1d` Implements a single iteration of a stencil computation (diffusion equation in one dimension) optimized by using shared memory. Each thread computes one element of the output array.

We did not concretize any of the parameters in programs, such as the number of threads and blocks, length of vectors, and size of matrices. Throughout the experiments, we set time limit to 1 second through Why3 API for each solver call (some of the solvers seem to run for one more second than the given time limit; we do not know the reason for this). We also set memory limit to 4000MB, but it seems that it is almost impossible to exhaust this amount of memory in a few

[5] `alt-ergo.lri.fr`, `www.cs.nyu.edu/acsys/cvc3`, `cvc4.cs.nyu.edu`, `www.eprover.org`, `z3.codeplex.com`.

[6] `https://fmt.ewi.utwente.nl/redmine/projects/vercors-verifier/wiki/Scp-examplepvl`

seconds. Experiments are conducted on a machine with two Intel Xeon processors E5-2670 (with eight cores, 2.6 GHz) and 128GB of main memory. The OCaml modules are compiled using `ocamlopt` version 4.03.0.

The result is summarized in Table 1. This table shows the performance of our method with and without the simplification introduced in Section 4 (shown in the second column). For the case where no simplification is applied, we have provided triggers that would help solvers finding an instance used in ElimAssign and rewriting (such as $b_2(n)$ in (1) and $i_2(t)$ in $\forall t.0 \leq t < N \rightarrow i_2(t) = N \cdot l + t$). Most of the tasks consist of two subtasks, only one of which has to be proved, and others are not divided into subtasks. The size of a VC is defined by the sum of the size of all formulas in it, and the size of a formula is the number of nodes in its abstract syntax tree. The number of tasks changes when the simplification is enabled. Usually the number increases because simplification may split a task into smaller ones, but it also decreases because a task is removed from VC if it is reduced to a trivial one.

Our implementation, with the simplification, successfully verified realistic GPU kernels, whereas it could not verify any of the programs without simplification. We also ran SMT solvers with one hour of time limit on each task before simplification, and confirmed that the numbers of proved tasks did not change. These results show that our simplification strategy is indeed effective. As mentioned in Section 4.4, our implementation tries two strategies of simplification: one of them (which we call S_1) applies MergeQuant and SimpAffine in this order, and the other (which we call S_2) applies only SimpAffine. We observed that both of S_1 and S_2 are needed to verify all the sample programs we examined in the experiments. We confirmed that (1) `vectorAdd` can be verified by using S_1, but cannot if only S_2 is used, whereas (2) `matrixMul2` can be verified by using S_2, but cannot if only S_1 is used.

The result also suggests a limitation of our current implementation. As we can see from the VC generation time and size of several programs, our method occasionally generates quite large VC, which is time- and memory-consuming to generate. The size of VC is sensitive to two factors: the number of assignments contains in the program, and the number of occurrences of program variables in the postcondition (they apply to `matrixMul4` and `diffusion1d`, respectively). In both cases, the increase of VC size appears to be caused mainly by iterated applications of ElimAssign which, in the worst case, almost doubles the size of the formula every time. We expect that the generation time can be reduced by further optimization, because during ElimAssign many redundant formulas are generated, and removed afterwards (indeed, in the case of `diffusion1d`, the output of ElimAssign has size approximately 1.06×10^7, which is nearly 60 times larger than the final VC).

6 Related Work

Functional correctness of GPU programs. Some of the existing tools support functional correctness verification by assertion checking or equivalence checking. PUG [19] and GKLEE [21] support assertion checking (as well as detecting other defects such as data races), but they cannot verify fully parameterized programs. Both of them require the user to specify the number of threads, and they duplicate each instruction by the specified number of threads to simulate lockstep behavior as a sequential program. PUG$_{para}$ [20] supports equivalence checking of two param-

Table 1 The number of proved/generated tasks, time spent for VC generation and SMT solving (sec), and size of VC, with and without VC simplification. LOC excludes blank lines and annotations. VC generation and SMT solving time is the average of ten executions. Result for `matrixMul4` without simplification was 23 for eight out of ten executions, and 24 for the other two.

program	simplify	result	VC gen.	SMT solving	VC size
`scp`	Y	1/1	0.0616	0.2598	2524
(6 LOC)	N	1/2	0.0033	5.0072	2663
`rotate`	Y	1/1	0.0729	0.2794	2603
(7 LOC)	N	1/3	0.0051	14.2263	4179
`vectorAdd`	Y	5/5	0.2305	0.8304	13291
(9 LOC)	N	3/7	0.0093	26.3784	9879
`matrixMul`	Y	17/17	1.4402	2.7105	62531
(29 LOC)	N	15/17	0.0532	13.0309	38137
`matrixMul2`	Y	34/34	15.0716	8.4726	160146
(34 LOC)	N	18/21	0.0739	18.1891	56623
`matrixMul4`	Y	158/158	1762.7441	36.5927	999572
(42 LOC)	N	{23,24}/29	0.1202	41.7131	103435
`diffusion1d`	Y	62/62	10928.0448	23.1609	176202
(20 LOC)	N	1/4	0.0097	14.3308	6511

eterized programs. They report results on equivalence checking of unoptimized and optimized kernels; equivalence checking of a parameterized matrix-transpose program resulted in timeout, so they had to concretize some of the variables.

Deductive approaches to functional correctness. Regarding deductive verification of GPU programs, two approaches have been proposed. Kojima and Igarashi adapted the standard Hoare Logic to GPU programs [13, 14]. Our work is based on theirs, although we do not use their inference rules as they are. Blom, Huisman and Mihelčić applied permission-based separation logic to GPU programs [3]. Their logic is implemented in the VerCors tool set,[7] and mechanization of their logic in proof assistant Coq is also studied [1]. Their approach can reason about race-freedom, in addition to functional correctness, by making use of the notion of permission, but it requires more annotations than ours.

Automated race checking. Race checking is one of the subject intensively studied in verification of GPU programs, and many tools have been developed so far [19, 7, 20, 22, 23, 2]. Although they use SMT solvers, their encoding methods for race-checking are different from ours in several ways. In particular, it is not necessary to consider all threads at a time, but only two threads suffice. This is because if there is a race, then there has to be a pair of threads that are to perform conflicting read/write (this is an important observation for optimization which, to our knowledge, first mentioned in [19] and detailed discussion on this technique is given in [2]). Therefore they model the behavior of a pair of threads (whose thread identifiers are parameterized), rather than all threads.

Reasoning about arrays. There is a technique to eliminate existential quantification over arrays, which is applied to the verification of C program involving arrays [16].

[7] Several examples are found at `https://fmt.ewi.utwente.nl/redmine/projects/vercors-verifier/wiki/Examples`.

Although we did not consider quantifier elimination over arrays explicitly, the effect of ElimAssign is similar to the quantifier elimination: if a variable a representing an intermediate value of some array and a does not appear in the postcondition, then we can regard a as an existentially quantified variable. Because ElimAssign removes a from the VC, it could be seen as a quantifier-elimination procedure. Further investigation on relationship to their idea and possibility of adapting it to our setting is left for future work.

7 Conclusion

We have presented an automated verification method of race-free GPGPU programs. Our method is based on symbolic execution and (manual) loop abstraction. In addition to the VC generation method, we have proposed several simplification methods that can help SMT solvers prove generated VCs. We have empirically confirmed that our method successfully verifies several realistic kernels without concretizing parameters and that the simplification method is effective for improving efficiency of the verification procedure. We expect that it is a feasible approach to the verification of functional correctness to check race-freedom by using the existing tools first, and then verifying functional correctness by using our method.

Another possible approach to the verification problem is to translate a GPU program (and its specification) into a sequential program, and then apply existing tools. It would not be difficult to write a translation, because in our setting GPU programs are treated mostly as sequential program, except for assignments. Because an assignment in a GPU program corresponds to multiple assignments in a sequential program, it has to be translated into a loop over threads or other construct that operates on multiple indices. An advantage of this approach is that we do not need to implement a verification condition generator (translation from a program to formulas). However, we would still need simplification techniques similar to the ones presented in Section 4, because the translation does not change the nonlinearity of the VCs, from which one of the main difficulty arises. Detailed investigation of this approach is left for future work.

Automatically inferring loop invariants is one of the interesting and important problems left for future work. Various methods to generate invariants have been proposed in the literature [24, 17, 11, 6]. Although they mainly target sequential programs, we expect that they can be adapted to GPU programs. To our knowledge, there is no previous work on applying these invariant generation methods to GPU programs (GPUVerify [2] uses Houdini algorithm [9] to find invariants, and PUG [19] uses predefined set of syntactic rules that can automatically derive an invariant if the program fragment matches a common pattern).

As addressed in Section 4, better manipulation of nonlinear formulas is also important. One of the possible direction to further extend the current state of the research would be to investigate the relationship to decidable nonlinear extensions of linear arithmetic [5, 18]. Although we do not expect that all the VCs are expressed in such theories, it would be interesting if these theories and their decision procedures bring us a new insight into the manipulation of nonlinear VCs.

Improving the strategy of simplification on VCs is also vital for scalability of our verification method. As we have discussed in Section 5, our simplification

method sometimes produces extremely large VCs, or even fails to generate VCs in a reasonable amount of time. Also, there seems to be room for optimization in the ElimAssign procedure. We expect that optimizing this part greatly reduces the amount of time spent for verification, because ElimAssign is one of the most time-consuming part of our verification method.

Specifically, there are several cases where SimpAffine does not work directly, but it does work after applying a certain type of modification (roughly the converse of MergeQuant). For example, consider the following formula (this formula is related to the verification of `matrixMul2` in our experiments).

$$\forall x.0 \le x < 2ab \rightarrow \\ (\exists y.0 \le y < b \wedge 2ay \le x < 2ay + a) \vee (\exists y.0 \le y < b \wedge 2ay + a \le x < 2ay + 2a).$$

Although it cannot be simplified by SimpAffine, if we rewrite it by decomposing variable x into two parts, we obtain

$$\begin{aligned} \forall x_1, x_2.0 \le x_1 < 2b \wedge 0 \le x_2 < a \rightarrow \\ (\exists y.0 \le y < b \wedge 2ay \le ax_1 + x_2 < 2ay + a) \vee \\ (\exists y.0 \le y < b \wedge 2ay + a \le ax_1 + x_2 < 2ay + 2a). \end{aligned}$$

Then, assuming $a, b > 0$, SimpAffine can rewrite $2ay \le ax_1 + x_2$ and $ax_1 + x_2 < 2ay + a$ into $2y \le x_1$ and $x_1 < 2y + 1$, respectively, and thus the first disjunct becomes $\exists y.0 \le y < b \wedge 2y \le x_1 < 2y + 1$. By rewriting the second disjunct similarly, we obtain a linear formula equivalent to the original one:

$$\forall x_1, x_2.0 \le x_1 < 2b \wedge 0 \le x_2 < a \rightarrow \\ (\exists y.0 \le y < b \wedge 2y \le x_1 < 2y + 1) \vee (\exists y.0 \le y < b \wedge 2y + 1 \le x_1 < 2y + 2).$$

At the time of writing, we do not have a concrete algorithm to decompose a quantifier as above. The current implementation requires users to write a specification to which SimpAffine works. We are currently working on automatically decomposing quantifiers before applying SimpAffine.

References

1. Asakura, I., Masuhara, H., Aotani, T.: Proof of soundness of concurrent separation logic for GPGPU in Coq. Journal of Information Processing **24**(1), 132–140 (2016)
2. Betts, A., Chong, N., Donaldson, A.F., Ketema, J., Qadeer, S., Thomson, P., Wickerson, J.: The design and implementation of a verification technique for GPU kernels. ACM Trans. Program. Lang. Syst. **37**(3), 10:1–10:49 (2015). DOI 10.1145/2743017. URL `http://doi.acm.org/10.1145/2743017`
3. Blom, S., Huisman, M., Mihelčić, M.: Specification and verification of GPGPU programs. Science of Computer Programming **95**(3), 376–388 (2014)
4. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd Your Herd of Provers. In: Boogie 2011: 1st Intl. Workshop on Intermediate Verification Languages, pp. 53–64. Wroclaw, Poland (2011). URL `https://hal.inria.fr/hal-00790310`
5. Bozga, M., Iosif, R.: On decidability within the arithmetic of addition and divisibility. In: V. Sassone (ed.) Proc. of FOSSACS 2005, *Springer LNCS*, vol. 3441, pp. 425–439. Springer (2005). DOI 10.1007/978-3-540-31982-5_27. URL `http://dx.doi.org/10.1007/978-3-540-31982-5_27`

6. Cachera, D., Jensen, T.P., Jobin, A., Kirchner, F.: Inference of polynomial invariants for imperative programs: A farewell to Gröbner bases. Science of Computer Programs **93**, 89–109 (2014). DOI 10.1016/j.scico.2014.02.028. URL http://dx.doi.org/10.1016/j.scico.2014.02.028
7. Collingbourne, P., Cadar, C., Kelly, P.H.: Symbolic testing of OpenCL code. In: K. Eder, J.a. Lourenço, O. Shehory (eds.) Proc. of Hardware and Software: Verification and Testing, *Springer LNCS*, vol. 7261, pp. 203–218. Springer Verlag (2012). DOI 10.1007/978-3-642-34188-5_18
8. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3), 365–473 (2005). DOI 10.1145/1066100.1066102. URL http://doi.acm.org/10.1145/1066100.1066102
9. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: J.N. Oliveira, P. Zave (eds.) Proc. of International Symposium of Formal Methods Europe (FME 2001), *Springer LNCS*, vol. 2021, pp. 500–517. Springer (2001). DOI 10.1007/3-540-45251-6_29. URL http://dx.doi.org/10.1007/3-540-45251-6_29
10. Flanagan, C., Saxe, J.B.: Avoiding exponential explosion: Generating compact verification conditions. In: Proc. of ACM POPL, POPL '01, pp. 193–205. ACM, New York, NY, USA (2001). DOI 10.1145/360204.360220. URL http://doi.acm.org/10.1145/360204.360220
11. Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: A robust framework for learning invariants. In: A. Biere, R. Bloem (eds.) Proc. of 26th International Conference on Computer Aided Verification (CAV 2014), *Springer LNCS*, vol. 8559, pp. 69–87. Springer (2014). DOI 10.1007/978-3-319-08867-9_5. URL http://dx.doi.org/10.1007/978-3-319-08867-9_5
12. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976). DOI 10.1145/360248.360252. URL http://doi.acm.org/10.1145/360248.360252
13. Kojima, K., Igarashi, A.: A Hoare Logic for SIMT programs. In: C. chieh Shan (ed.) Proc. of Asian Symposium on Programming Languages and Systems (APLAS 2013), *Springer LNCS*, vol. 8301, pp. 58–73 (2013)
14. Kojima, K., Igarashi, A.: A Hoare logic for GPU kernels. ACM Transactions on Computational Logic (2016). To appear. A revised and extended version of [13].
15. Kojima, K., Imanishi, A., Igarashi, A.: Automated verification of functional correctness of race-free GPU programs. In: S. Blazy, M. Chechik (eds.) Verified Software. Theories, Tools, and Experiments - 8th International Conference, VSTTE 2016, Toronto, ON, Canada, July 17-18, 2016, Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 9971, pp. 90–106 (2016). DOI 10.1007/978-3-319-48869-1_7. URL http://dx.doi.org/10.1007/978-3-319-48869-1_7
16. Komuravelli, A., Bjørner, N., Gurfinkel, A., McMillan, K.L.: Compositional verification of procedural programs using Horn clauses over integers and arrays. In: R. Kaivola, T. Wahl (eds.) Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015., pp. 89–96. IEEE (2015)
17. Kovács, L., Voronkov, A.: Finding loop invariants for programs over arrays using a theorem prover. In: M. Chechik, M. Wirsing (eds.) Fundamental Approaches to Software Engineering, *Springer LNCS*, vol. 5503, pp. 470–485. Springer Berlin Heidelberg (2009). DOI 10.1007/978-3-642-00593-0_33. URL http://dx.doi.org/10.1007/978-3-642-00593-0_33
18. Lechner, A., Ouaknine, J., Worrell, J.: On the complexity of linear arithmetic with divisibility. In: Proc. of 30th Annual ACM/IEEE Symposium on Logic in Computer Science, (LICS 2015), pp. 667–676. IEEE (2015). DOI 10.1109/LICS.2015.67. URL http://dx.doi.org/10.1109/LICS.2015.67
19. Li, G., Gopalakrishnan, G.: Scalable SMT-based verification of GPU kernel functions. In: Proc. of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10), pp. 187–196. ACM (2010). DOI 10.1145/1882291.1882320
20. Li, G., Gopalakrishnan, G.: Parameterized verification of GPU kernel programs. In: IPDPS Workshop on Multicore and GPU Programming Models, Languages and Compilers Workshop, pp. 2450–2459. IEEE (2012)
21. Li, G., Li, P., Sawaya, G., Gopalakrishnan, G., Ghosh, I., Rajan, S.P.: GKLEE: concolic verification and test generation for GPUs. In: J. Ramanujam, P. Sadayappan (eds.) Proc. of ACM PPoPP, pp. 215–224. ACM (2012). DOI 10.1145/2145816.2145844. URL http://doi.acm.org/10.1145/2145816.2145844
22. Li, P., Li, G., Gopalakrishnan, G.: Parametric flows: automated behavior equivalencing for symbolic analysis of races in CUDA programs. In: Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12). IEEE Computer Society Press (2012)

23. Li, P., Li, G., Gopalakrishnan, G.: Practical symbolic race checking of GPU programs. In: T. Damkroger, J. Dongarra (eds.) Proc. of Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC 2014), pp. 179–190. IEEE (2014). DOI 10.1109/SC.2014.20. URL http://dx.doi.org/10.1109/SC.2014.20
24. McMillan, K.: Quantified invariant generation using an interpolating saturation prover. In: C. Ramakrishnan, J. Rehof (eds.) Tools and Algorithms for the Construction and Analysis of Systems, *Springer LNCS*, vol. 4963, pp. 413–427. Springer Berlin Heidelberg (2008). DOI 10.1007/978-3-540-78800-3_31. URL http://dx.doi.org/10.1007/978-3-540-78800-3_31
25. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: intermediate language and tools for analysis and transformation of C programs. In: Proc. of 11th Intl. Conf. on Compiler Construction (CC 2002), *Springer LNCS*, vol. 2304, pp. 213–228 (2002). DOI 10.1007/3-540-45937-5_16. URL http://dx.doi.org/10.1007/3-540-45937-5_16
26. Nguyen, H.: GPU Gems 3, first edn. Addison-Wesley Professional (2007). http://developer.nvidia.com/object/gpu-gems-3.html
27. NVIDIA: NVIDIA CUDA C Programming Guide (2014). URL http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html