Analysis of GPGPU Programs for Data-race and Barrier Divergence

Santonu Sarkar[1], Prateek Kandelwal[2], Soumyadip Bandyopadhyay[3] and Holger Giese[3]
[1]*ABB Corporate Research, India*
[2]*MathWorks, India*
[3]*Hasso Plattner Institute für Digital Engineering gGmbH, Germany*

Keywords: Verification, SMT Solver, CUDA, GPGPU, Data Races, Barrier Divergence.

Abstract: Todays business and scientific applications have a high computing demand due to the increasing data size and the demand for responsiveness. Many such applications have a high degree of parallelism and GPGPUs emerge as a fit candidate for the demand. GPGPUs can offer an extremely high degree of data parallelism owing to its architecture that has many computing cores. However, unless the programs written to exploit the architecture are correct, the potential gain in performance cannot be achieved. In this paper, we focus on the two important properties of the programs written for GPGPUs, namely i) the data-race conditions and ii) the barrier divergence. We present a technique to identify the existence of these properties in a CUDA program using a static property verification method. The proposed approach can be utilized in tandem with normal application development process to help the programmer to remove the bugs that can have an impact on the performance and improve the safety of a CUDA program.

1 INTRODUCTION

With the order of magnitude increase in computing demand in the business and scientific applications, developers are more and more inclined towards massively data-parallel computing platform like a general-purpose graphics processing unit (GPGPU) (Nickolls and Dally, 2010). GPGPUs are programmable co-processors where designers can develop applications in CUDA if they are using NVIDIA GPGPU. CUDA (Compute Unified Device Architecture) is an extension to standard ANSI C through additional keywords. A CUDA program follows a Single-Program Multiple- Data (SPMD) model, where a piece of code, called *kernel*, is executed by hundreds of threads depending on the device's computing capability, in a lock-step fashion (where all the participating threads execute the same instruction). The developer can bundle these threads into a set of thread-blocks, where a set of threads in a block can share their data and synchronize their actions through a built-in barrier function called `__syncthreads()`.

While sharing of data among threads is essential for any reasonably complex parallel application, it can lead the program to a *unsafe* state where the behavior of the program is unpredictable and incorrect. The safety property of a program, which informally means that the program will never end-up in an erroneous state, or will never stop functioning in an arbitrary manner, is a well-known and critical property that an operational system should exhibit (Lamport, 1977). A parallel program can reach an unsafe state when multiple threads enter into a race condition, conflict with each other for data access enters into a deadlock situation and so on. Such a situation never arises in a sequential program. Thus, unlike a sequential software, detecting a violation of the safety property and identifying its cause in a parallel program is far more challenging. Petri net based verification of parallel programs has been reported in (Bandyopadhyay et al., 2017; Bandyopadhyay et al., 2018), however, it checks only the computational equiavlence.

One of the well-known techniques for detecting such an unsafe operation in a program is the property verification method. This approach models potential causes for a program safety violation such as data-races, or deadlock as properties of the program. The verification program analyzes these properties, extracted from the code, for their satisfiability. While this technique has been proposed for a long time and used in the context of sequential programs, recently it has gained renewed interest with the resurgence of GPGPUs and its programming model. Analysis of a parallel program differs from the analysis of a se-

Sarkar, S., Kandelwal, P., Bandyopadhyay, S. and Giese, H.
Analysis of GPGPU Programs for Data-race and Barrier Divergence.
DOI: 10.5220/0006834904600471
In *Proceedings of the 13th International Conference on Software Technologies (ICSOFT 2018)*, pages 460-471
ISBN: 978-989-758-320-9

quential program because of the possible ordering of instructions, can be many (unless restricted otherwise with synchronization calls or atomic access) when these instructions are executed in parallel by multiple threads. The analysis of such a program requires the methodology to consider all possible inter-leavings of the instructions where there is no specific ordering imposed by the program. This has a significant impact on the number of permutations that need to be analyzed. This problem can be severe in the case of GPU programs, as the number of threads can be in the order of thousands, with absolutely no synchronization available between two threads belonging to separate blocks.

We propose a static verification tool that can analyze a CUDA program and verify all possible data-races spanning all the barrier intervals, which may occur when the program is executed on a GPU. The major contributions of this paper are as follows:

1. Existing literature such as (Li and Gopalakrishnan, 2010; Lv et al., 2011) detects a single data race or a set of races between two barrier functions. This, in turn, requires the developers to repeatedly run and fix the data race conditions, which can be quite cumbersome and time-consuming when the program is large, involving many barriers. Thus, our approach generates more comprehensive results in one run which can substantially improve the program repairing time.
2. Our method can detect whether the barriers are divergent.
3. We have provided a formal analysis of our data race and barrier divergence detection algorithms and proved that our approach is sound, i.e., it does not have any false positive results.

The paper has been organized as follows. In Section 3, we provide an overview of the approach. Next, we elaborate the program translation mechanism in Section 4. In Section 5 we elaborate the verification approach. We report the experimental study of our tool in Section 6. Finally, we conclude our paper.

2 RELATED WORK

Unlike a sequential program, a parallel program execution can lead to several new safety and correctness issues (Kirk and Hwu, 2016; Lin et al., 2015). A more recent article (Ernstsson et al., 2017) further substantiates several safety issues in GPU programming.

An early approach by Boyer et al. (Boyer et al., 2008) proposed a dynamic analysis of a CUDA program. The CUDA kernel is instrumented to generate runtime traces of memory accesses by different threads, which is analyzed to discover various bugs like data races and bank conflicts.In comparison, our approach analyzes the symbolic constraints associated with the potentially conflicting access, along with checking for divergent barriers, which are not handled in the former case.

PUG tool (Li, 2010) is a symbolic verifier based approach to detect data race conflicts, barrier divergence, and bank conflicts. PUGpara (Li and Gopalakrishnan, 2012) is an extension of the PUG tool which can also check for functional correctness and equivalence of CUDA kernels. PUGpara models a single parameterized thread and uses symbolic analysis for verification of data race and equivalence checking. A limitation of the tool is that the tool does not detect all the races in a program, and exits on the *first encounter* of a race. Also, the barrier divergence detection mechanism in presence of branches, checks if the number of barriers executed along the path to be the same in the absence of branch divergence, which is not correct as per the CUDA specification, where for each barrier synchronization call, it is expected that either all the threads within the block should execute it, or none at all.

Performance degradation analysis of GPUs concerning bank conflicts and coalesced global memory has been modeled as an unsatisfiability problem in (Lv et al., 2011). The approach produces a counter-example where threads with consecutive thread-ids may not access the consecutive memory locations. Here factors like data-type and the nature of access play an essential role (CUD, 2017). While the approach reported in the paper is novel, it is indeed not robust in considering all possible scenarios of un-coalesced access or complex bank conflict scenarios.

A combination of static and dynamic analysis has been proposed by (Zheng et al., 2011; Zheng et al., 2014) where, initially a static analysis is used to reduce the number of statements to be instrumented for checking data races. Next, the access patterns of the instrumented statements are collected and analyzed using dynamic analysis.

An approach presented in (Said et al., 2011) generates concrete thread schedules that can deterministically trigger a particular data race. This approach also leverages the satisfiability modulo theory (SMT) to generate the schedule, modeled as constraints. The approach, however, has been developed for task-parallelism in Java involving fork/join semantics.

Recently a synchronous delayed visibility semantics based program verification has been proposed(Betts et al., 2015) to detect data races and barrier divergence. Unlike this approach, we are using the notion of the predicated form of a thread in our approach while translating the program to assist in the generation of symbolic constraints.

Symbolic execution-based approach for detecting data races in OpenCL code has been explored in (Collingbourne et al., 2012) which records each memory access and then use this information to detect the data-race. Naturally, approach is computationally more expensive than an efficient SMT based solver like ours.

Formal Semantics for heterogeneous CUDA C is presented in (Hathhorn et al., 2012) with an interpreter built upon the approach. With the help of the interpreter based execution, the developers can identify specific instances of race conditions and deadlocks due to barrier divergence.

The test amplification based approach is presented in (Leung et al., 2012), where it is shown that in certain conditions, the results obtained by a single test execution of a kernel can be amplified to prove the correctness properties like data-race freedom for the kernel in general during static analysis.

A concolic testing based technique is presented in GKLEE (Li et al., 2012b; Li et al., 2012a) which looks into both correctness and various performance issues of GPU programs.

Researchers have used finite state model-based analysis to detect occurrences of conflicts among atomic synchronization commands in (Chiang et al., 2013) which can help in generating alternate delay based scheduling of GPU programs.

The concept of barrier invariants is presented in (Chong et al., 2013) to enable verification for data dependent kernels, specially when the data are in the shared memory of a GPU. However, the approach does not directly address the data race situation.

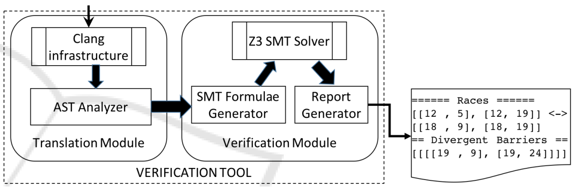
In (Collingbourne et al., 2013), a lock-step semantics for GPU kernels has been proposed to deal with unstructured GPU kernels.

Permission-based separation logic presented in (Huisman and Mihelčić, 2013; Blom and Huisman, 2014), has been used for verification of OpenCL programs. Here, the program is annotated by read and write permissions, where at most one thread can hold write permission to a location, and multiple threads can hold read permission simultaneously.

Bardsley et al.(Bardsley and Donaldson, 2014) presented a method to analyze atomics and warp s synchronization for GPU kernels without explicit barriers using a two-pass and a re-sync approach. The approach has been proposed to analyze the cases of inter and intra-warp execution.

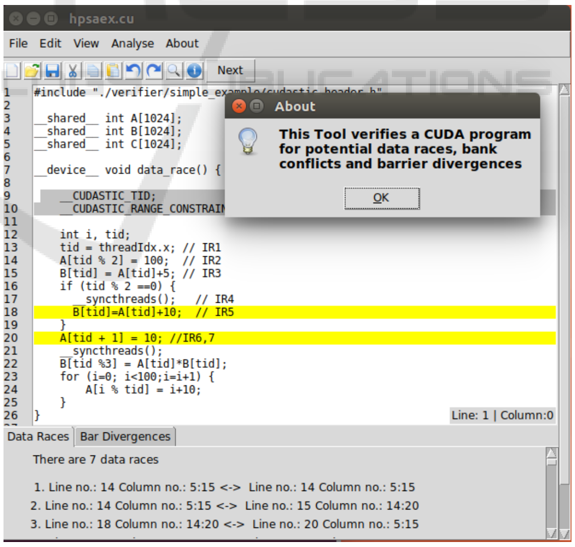
The problem of optimal barrier in SPMD and multi-threaded programs has been studied in (Stöhr and O'Boyle, 1997; Oboyle et al., 1995; Darte and Schreiber, 2005; Dhok et al., 2015), where it has been shown that the problem of identifying the optimal placement of a barrier is tractable ensuring that the problem can be solved for practical purposes.

```
__shared__ int A[1024],int B[1024],
           int C[1024];
__device__ void data_race() {
    int i, tid;
    tid = threadIdx.x;// IR 1.
    A[tid % 2]= 100;    // IR 2.
    B[tid]= A[tid]+5;   // IR 3.
    if ((tid % 2 ==0) {
      __syncthreads();   // IR 4.
      B[tid]= A[tid]+10;// IR 5.
    }
    A[tid+1] = 10;//IR 6, IR 7
    __syncthreads();
    B[tid %3] = A[tid]*B[tid];
    for (i=0; i<100; i++) { //IR 8
      A[i % tid] = i+10;   // IR 9
    }
}
```



```
====== Races ======
[[12 , 5], [12, 19]] <->
[[18 , 9], [18, 19]]
== Divergent Barriers ==
[[[[19 , 9], [19, 24]]]]
```

Figure 1: A Schematic Diagram of Our tool.



This Tool verifies a CUDA program for potential data races, bank conflicts and barrier divergences

```
1  #include "./verifier/simple_example/cudastic_header.h"
2
3  __shared__ int A[1024];
4  __shared__ int B[1024];
5  __shared__ int C[1024];
6
7  __device__ void data_race() {
8
9      __CUDASTIC_TID;
10     __CUDASTIC_RANGE_CONSTRAIN
11
12     int i, tid;
13     tid = threadIdx.x; // IR1
14     A[tid % 2] = 100;  // IR2
15     B[tid] = A[tid]+5; // IR3
16     if (tid % 2 ==0) {
17       __syncthreads();   // IR4
18       B[tid]=A[tid]+10;  // IR5
19     }
20     A[tid + 1] = 10; //IR6,7
21     __syncthreads();
22     B[tid %3] = A[tid]*B[tid];
23     for (i=0; i<100;i=i+1) {
24         A[i % tid] = i+10;
25     }
26 }
```

There are 7 data races

1. Line no.: 14 Column no.: 5:15 <-> Line no.: 14 Column no.: 5:15
2. Line no.: 14 Column no.: 5:15 <-> Line no.: 15 Column no.: 14:20
3. Line no.: 18 Column no.: 14:20 <-> Line no.: 20 Column no.: 5:15

Figure 2: GUI of the tool.

3 PROPOSED APPROACH

Here we translate a CUDA kernel program into a set of well-formed logical formulae (WFF) in Static Single Assignment (SSA) form and then perform constraint solving using an SMT solver to check for correctness properties as a part of the symbolic execution. Our approach is based on symbolic analysis, where once a CUDA kernel program is translated into a set of logical well-formed formulas, we use them as constraints for a generic thread. We then use the concept of two threaded execution and generate appropriate additional constraints that should hold if a given operation has a data race. These constraints are then fed to the Z3 SMT solver [1] to see if it can find a satisfying assignment to the constraints, if yes, then the tool returns the assignments as a witness for the data race condition, else reports the inability in verifying the data race condition. Our approach aims to detect all the pairs of statements that can lead to data race and all the instances of the barriers where the barrier divergence is possible. The upper part of Figure 1 shows a sample CUDA code which we use as a running example. The lower part of the figure is a schematic diagram of our approach comprising of two modules, the translation, and the verification module. The translation module uses CLANGs (Lattner and Adve, 2004) Abstract Syntax Tree (AST) to convert the CUDA kernel source file to a set of SMT equations. The Verification module takes the translated set and creates the SMT equations encoded with the properties that need to be verified. The satisfiability for these equations is checked using the SMT Solver, where a witness is returned if the equations are satisfiable and imply the presence of data races or barrier divergences. All such violations of correctness properties are collected and reported back to the user along with their location in the source code. Figure 2 shows the GUI of the tool that highlights the statements having potential data races and barrier divergence.

Example 1. *Let us consider the sample CUDA kernel shown in Figure 1, that uses three arrays shared by multiple threads. Since each thread (with unique thread id threadIdx.x) accesses one element of the array A, threads with even thread-ids 0, 2, and 4 access the same shared memory location, resulting in a conflict in line with the comment "IR 2". Similarly, there is a chance of a read-write conflict between lines IR 2 and IR 3.*

The code has a barrier call in line 13 which ensures that all the threads are synchronized at the barrier before proceeding for the next statement. We can see that there can be a read-write conflict between two threads in line IR 5 and IR 6.

There is a barrier call in line IR 4 inside the if-condition that creates a barrier divergence since all threads do not reach the barrier since the barrier function is inside the `if` *condition. In the paper, we will illustrate only IR 5 and 6.*

[1]https://github.com/Z3Prover/z3

4 SOURCE PROGRAM TO WELL FORMED FORMULAE

Our translation module can handle most of the CUDA-C grammar necessary for writing kernels. In the current implementation, we are not handling general function calls besides the call to `__syncthreads()`, nor pointers and assignments.

4.1 Translation Rules

The translation module shown in Figure 1, uses CLang infrastructure to define user-defined semantic actions for each AST node. We have defined specific semantic actions that convert each source code statement of a CUDA kernel into a predicated execution form; formally represented as $predicate \rightarrow action$.

Here, both the predicate and the action are well-formed formulae generated using the translation rules. Before converting a statement into a WFF, the translation module converts each variable into its static single assignment (SSA) form. Hereinafter, we refer to the SSA form of a variable as the "augmented variable".

Assignments. For an assignment statement, we create an action as ($==$ $augmented\_variable\ lvalue$) and return to the parent AST node. If there are multiple declaration and assignments such as `int x = 10, y = 10;`, the translator will create action expression for each of them.

Important Unary and Binary Operators. The tool supports arithmetic and logical operators such as the not, "!", the bitwise inversion "¬", unary minus "-", incr and decr (both postfix and prefix), "++" and "–" operators. The parser also allows the usual binary arithmetic and logical operators. In this context, we highlight a special case of the assignment operator, "=". Here the translation process ensures that the SSA index of the *lvalue* of the assignment expression is incremented.

Array Expression. Consider the expression `A[tid % 2]` in line 4, in Example 1 of the CUDA program. For this array variable, the translation mechanism needs to consider the variable and the index expression that points to a specific element in the array. When multiple threads access different elements of an array simultaneously, there can be a possibility of access conflict. The translator module creates a well-formed formula combining the array variable (in SSA form) and the index expression to detect such a potential conflict. Here we make use of McCarthy's array access (select-store) axiom (McCarthy, 1962; Manna, 1974) to construct the formula. We explain this later towards the end of the section.

Conditional Statements. Conditional statements of a CUDA program are tricky. Here, the translation of an if and if-else statement to a WFF must consider the fact that multiple threads of a GPGPU can take different paths during the execution of the CUDA kernel. Since we are using SSA representation, the SSA indices might have different values in the `then` and the `else` blocks. This is a classic compiler translation problem, which is handled by creating a ϕ statement which acts as a multiplexer to select the appropriate augmented variable. Furthermore, a conditional statement will have a predicated execution, where the predicate is created from the conditional expression. The idea is illustrated below. Note that SSA indices of the variable `y` change inside the if block.

CUDA program fragment	Predicated form
`int x = 10;`	$(true) \implies x1 = 10$
`int y = 4;`	$(true) \implies y1 = 4$
`if ( x > 9 ) {`	$(x1 > 9) \implies y2 = 6$
`    y = 6;`	$!(x1 > 9) \implies y2 = y1$
`}`	

For Loop. A CUDA kernel can have a for-loop, which implies that each thread will execute the for-loop. The translator model allows a simplified for loop construct with an initialization expression, the guard condition and the increment operation, of the form "**var = var op k**", where **k** is a constant value or an expression that does not change over iterations, and **op** is one of the `+`, `-`, `<<`, `>>` operations. We construct predicates to model an arbitrary and valid iteration of the loop out of the three expressions and use it for the statements inside the loop body.

CUDA program fragment	Predicated form
`int y=0;`	$(true) \implies y0 = 0$
`for (int i=0;`	$(true) \implies i1 = 0$
`  i<100; i++) {`	$(true) \implies i2 = (i1 + i2k \times 1) \wedge (i2k \geq 0)$
`    y = i+9; }`	$(i2 < 100) \implies y1 = i2 + 9$

In this particular illustration, 'i2' represents the value of the index 'i' at an arbitrary iteration of the loop. Note that the value of 'i2' depends on a free variable 'i2k', which models an arbitrary kZ^{th} iteration. Since the iteration is modeled using a free variable, the WFF allows modeling of conflict scenarios where different threads execute different iterations of the loop.

4.2 Translated Well Formed Formulae (WFF)

The program statements are converted into a set of SMT compliant well formed formulae, which we can express in Backus Naur Form as:

expr := **num** | **var** | (**unaryop** expr) | (**binop** expr1 expr2)
| (`store` **var** expr1 expr2) | (`select` **var** expr)

Here **var** is a variable, **num** $\in \mathbb{Z}$, **unaryop**$\in$ {`!`, `~`} and **binaryop** $\in$ {`=`, `+`, `-`, `*`, `/`, `%`, `&&`, `||`, `^`, `&`, `|`}.

The `select` operation is used to read a value at an index i from array variable. The expression `(select A i)` corresponds to the value stored in the array `A` at position `i`.

The `store` operation is used to express a writing a value `v` in an array at a particular index `i`. For example, `A[i] = v` is converted to an expression `(store A i v)`.

Example 2. *Consider the code snippet shown in Figure 1. The translator module will create a set of WFFs. For brevity, we show the translated WFF for the lines annotated with IR 4, 5, and 6.*

```
4) ('==', ('%','tid4132775177_1','2'), '0')
   => BAR
5) ('==', ('%','tid4132775177_1','2'), '0')
   => ('==', 'B1306833491_2', ('store',
   'B1306833491_1', 'tid4132775177_1',
   ('+', ('select', 'A1271100794_1',
   'tid4132775177_1'), '10')))
6) ('==', ('%','tid4132775177_1','1'), '0')
   =>('==', 'A1271100794_2',('store',
  'A1271100794_1',('+', 'tid4132775177_1',
  '1'), '10'))
```

Now consider the body of the for-loop (IR 8 and 9). The translator module will create a predicated form for the loop and the loop body.

```
8) True => ('and', ('>=','i2473756790_2_k', 0),
   ('==','i2473756790_2',('+',
   'i2473756790_1',('*','i2473756790_2_k',
   '1'))))
9) ('and', 'True',('<','i2473756790_2',
   '100'))=> ('==', 'A1271100794_3',
   ('store','A1271100794_2',('%',
   'i2473756790_2','tid4293592209_1'),
   (+ i2473756790_2, '10')))
```

As discussed earlier, since i2 represents a valid value of the loop index variable 'i' at an arbitrary loop iteration, the generated WFF considers all the possible cases across the array. Thus, the generated WFF#8 means that 'i2' can take any value as long as there exists a valid 'i2k', and in conjunction with WFF#9, the value of ''i2k' is bounded such that we deal with only valid iterations.

5 VERIFICATION METHOD

We have decoupled the verification module from the translator module to provide greater flexibility and extensibility to the verification module. For instance, one can add multiple correctness checking procedures working on the same formulas with the same or different degree of granularity if such a need arises. As a concrete example, we intend to introduce *intra-warp bank conflicts* detection module as a future extension of our current tool.

The basic workflow of the verification process involves taking translated statements from the translation module as input, along with the *GlobalVariableMap*. The next step is to generate statements corresponding to a single, generic threaded execution. A two threaded execution model is then created using this generic threaded statement. Here we would like to highlight a crucial distinction between our approach from the general symbolic execution approach. In our approach, we are translating the program to generate a set of constraints, and these constraints are solved for verification of a correctness property. Like a symbolic execution approach, these constraints are generated by treating input as symbolic values; however, unlike a symbolic execution, we do not define any operational semantics as far as execution of the statement is concerned. This makes our approach *more lightweight* that a full-fledged operational semantics driven approach. In our approach, we iterate over each pair of possibly conflicting statements and test it for the correctness by considering all the necessary constraints on which the statements depend.

We now define three properties that our verification method is based on.

Property 1. If a SMT solver generates a model M for a Q of WFFs then $M \models Q$. This property implies the following. Let P be a program, and P is translated into the set M of well-formed formula (WFF). This set of WFF represents the model (of the program) given to the SMT solver.

Let Q be the set of properties that we want to verify. In our case, the set Q contains both the data race and barrier divergent property which is in the form of the set of WFF. This property means that if M satisfies Q, then the program contains both data race and barrier divergence.

Property 2. A kernel is free of data races iff any two arbitrary threads of execution of the kernel are free of data races.

Property 3. A kernel is free of barrier divergence if, for any two threads in the block, an arbitrary pair of thread do not diverge.

The entire verification approach follows 2-threaded execution model, where to verify a correctness property, the verifier module considers a pair of arbitrary threads. This approach is valid because the correctness properties like data races and barrier divergence are pair-wise properties, that is, in a given set of threads, unless all the pairs of threads can be proved to behave correctly, the correctness property of the entire program cannot be proved. Therefore, if one wants to verify the program for the absence of data race and barrier divergence, considering an arbitrary pair of distinct threads and checking if they violate the correctness, suffices the purpose.

5.1 Data Race Detection

The verification module handles the verification process of different correctness properties. As mentioned earlier, this process takes a set of translated statements and the GlobalVariableMap as input from the translator module. The verification process first converts each translated statement into a *threaded form* where each variable which is not *global* or *shared*, is replaced with a thread local variable. This ensures that during the two threaded analysis, the local variables do not affect other variables due to their respective constraints. This process is done by recursively visiting each expression in the well-formed formula and converting it to its corresponding threaded version. The function that converts translate statements into a threaded form is named as `ThreadIFY()` in this paper.

Once the threaded version is generated, the verification module creates thread specific copies for two symbolically parameterized threads, t_i and t_j which are used in verification of correctness properties, discussed in the following subsections.

*Condition for data race:*A data race occurs when two distinct threads access the same location such that one of the accesses is a write operation and these two accesses are within the same barrier interval. It may be noted that a barrier interval is the sequence of program statements between two barrier synchronization calls.

Barrier Synchronization: CUDA provides the barrier synchronization as the only synchronization primitive. The underlying principle of a barrier synchronization is that a thread upon encountering a barrier call waits until all the threads in the thread block reach the barrier, after which they resume their execution. Barriers ensure that all the changes done to the shared memory space are registered. The reason why access to the same location results in a data race is that the CUDA does not provide any guarantee about when the changes to a shared location will be reflected within a barrier interval, also, since multiple threads are accessing the same location, even if one guarantees that the changes will be reflected immediately, the access can follow a different order, which can result in an inconsistency in the value read by different threads. Consider the code snippet shown in Example 1. The translator module will create the following set of WFF. There are auxiliary data structures that are created in this stage too, which, for the sake of brevity, are not described here. Here we show the translated WFF for the lines annotated with IR 1 to IR 7.

Encoding Data race: Let us consider the two threaded execution model where two threads t_i and t_j encounters a data race, in a pair of statements, and the operations that conflict, involve conflicting access such as `A[tid] = 100`, and `y =A[tid+10] + y`. Let us assume without loss of generality that the first access is a write access, taking place in the thread t_i and the second access is a read access taking place in another thread t_j , also, let us assume that the variable `y` in the second access is a local variable, hence the accesses to the array, A are the only conflicting operations in this example. A data race will occur if the location $A + tid_i$ and $A + tid_j + 10$ are same, provided that $tid_i \neq tid_j$. Hence, there condition for checking data race roughly translates to, $tid_j \neq tid_i \wedge tid_i == (tid_j + 10)$. Feeding this to the constraint solver, we are likely to get a satisfying model where $tid_i = tid_j + 10$. Since an operation is a part of a statement, and each statement is in a predicated form, we need to also make sure that:

a) the predicate evaluates to `true` for both of the statements, otherwise a data race cannot take place,

b) since the operations in the current statement might involve certain symbols that are defined in preceding statements, we need to include them as well. Let us represent the predicates defined in the preceding statements are P_i and P_j for the thread t_i and t_j respectively. The verification condition then checks for the satisfiability of the following equation (Φ) where, $\Phi : (tid_i \neq tid_j) \wedge P_i \wedge P_j \wedge (tid_i == tid_j + 10)$. More generally, assuming that the conflicting operations involve accesses on indices $index_i$ and $index_j$, the data race condition becomes $\Phi : (tid_i \neq tid_j) \wedge P_i \wedge P_j \wedge (index_i == index_j)$. *If Φ has a satisfying assignment, the data race can be confirmed.* However, if the SMT solver is not able to find any satisfying assignment, it may be due to the solver time-out. In such a case, one can only state that the verification module was not able to prove the existence of data race, rather than saying that there is no data race. This is commonly known as the *soundness property* of the algorithm, which we prove in Section 5.3.2. The pseudo code for data race detection algorithm is given in Algorithm 1.

Algorithm 1: **DataraceDetection** (Translated-statements).

```
1: raceing-statement=$\emptyset$;
2: conflicting-statements = $\{\langle s_i, s_j\rangle \mid s_i, s_j \in$ Translated-statement $\wedge s_i$ conflicts with $s_j\}$
3: thread1 = ThreadIFY(Translated-statements,1)
4: thread2 = ThreadIFY(Translated-statements,2)
5: for $\langle s_l, s_m\rangle \in$ conflicting-statements do
6:     conflicting-op= $\{\langle op_i^l, op_j^m\rangle \mid op_i^l \in theread1.s_i, op_j^m \in thread2.s_j$ and $op_i^l$ conflicts with $op_j^m\}$
7:     for $\langle op_i^l, op_j^m\rangle \in$ conflicting-op do
8:         smt-eqn = Generate-SMT-EQN ($op_i^l, op_j^m$,datarace)
9:         if CHECK-SAT(sat-eqn)==SAT then
10:            raceing-statement = raceing-statement $\cup \langle op_i^l, op_j^m\rangle$
11:        end if
12:        conflicting-op=conflicting-op $\setminus\langle op_i^l, op_j^m\rangle$
13:    end for
14:    conflicting-statements= conflicting-statements$\setminus\langle s_l, s_m\rangle$
15: end for
16: return racing-statement
```

5.2 Barrier Divergence Detection

Barrier divergence occurs when for a barrier some threads in a block execute the call, while others do not. This can happen in cases when a barrier call is executed only in some scenarios. For instance, when a barrier is present within a conditional statement, the condition might not hold true for some threads. CUDAs programming guide specifies that the behavior in such scenarios is not defined. It is very likely that such a call might result in a deadlock, and hence can impact the correctness of the program. We need to verify if it is possible that for a given barrier synchronization call, one thread will execute it while others will not, if yes, then we have a possible barrier divergence. Since we are tracking the predicates for each statement separately, drawing the terminology from the data race section, this gives us the equation Ψ to be satisfied where, $\Psi : (tid_i \neq tid_j) \wedge (P_i \oplus P_j)$ This means that for two different threads Ψ is true if either P_i or P_j is true. *If Ψ has a satisfying assignment, the current barrier call is divergent, and the model represents the witness to it.* However, if there is no satisfying assignment, as in the case for data race, here too, we cannot say that the current barrier indeed is free of divergence. We prove the soundness of the barrier divergence approach in Section 5.3.3. The pseudo code for barrier divergence detection is given in Algorithm 2.

Algorithm 2: **BarrierDivergence** (Translated-statements).

```
 1: diverging-barriers=$\emptyset$;
 2: barrier-list = $\{s_b \mid \forall s_b \in$ Translated-statement $\wedge s_b =$ __syncthreads() $\}$
 3: thread$_i$ = ThreadIFY(Translated-statements, $i$)
 4: thread$_j$ = ThreadIFY(Translated-statements, $j$)
 5: barrier-pairs=$\{\langle b_i, b_j\rangle \mid b$ is barrier-list and $b_i$ is $b$ in thread$_i$, $b_j$ is b in thread$_j\}$
 6: for $\langle b_i, b_j\rangle \in$ barrier-pairs do
 7:     smt-eqn = Generate-SMT-EQN ($b_i, b_j$,barrier)
 8:     if CHECK-SAT(sat-eqn)==SAT then
 9:         diverging-barriers = diverging-barriers $\cup$ $\langle op_i^l, op_j^m\rangle$
10:     end if
11:     barrier-pairs=barrier-pairs$\setminus\langle b_i, b_j\rangle$
12: end for
13: return diverging-barriers
```

Example 3. *Consider the CUDA kernel shown in Figure 1, Example 1 and the set of WFF in Example 2. These WFF represent the model of the program which is verified against the data race and barrier divergence conditions. Let us consider the data race detection. The verification module detects a data race condition for various conflicting operations by creating two-threaded SMT equations. For instance, let us consider the conflict between IR 5. And IR 6. The main clause to encode the data race condition for the two threaded version will have an extra predicate for IR 6 (for the enclosing* `if` *condition), which needs to be true as well. The clause will look like:*

```
True AND (t_2__tid4132775177_1 % 1))==(0)
AND (t_1__tid4132775177_1 ==
(t_2__tid4132775177_1+1)
```

Reporting violations:
The generation of witnesses occur for that portion of the code where the tool is able to detect a data races or a barrier divergence. The verification module reports them along with the line number and column number for the instances of variables (line 18 and 20) as shown in Figure 1.

5.3 Formal Analysis of Our Approach

In this section, we discuss two important aspects of our approach, namely the termination of our algorithm and the soundness. We analyze these two aspects for both data-race as well as barrier divergence algorithms.

5.3.1 Soundness of Translation

Theorem 1. *If no satisfying assignment for the WFFs Φ_i and Φ_j of the two threads i and j and the condition linking to statements $\phi_{i,j}$ exists, then no execution of the two threads i and j leading to a situation fulfilling the condition linking to statements $\phi_{i,j}$ exists.*

Proof. Assuming that an execution of the two threads i and j leading to a situation fulfilling the condition linking to statements $\phi_{i,j}$ exists, we will show that then also a satisfying assignment for the well formed formulas Φ_i and Φ_j of the two threads i and j and the condition linking to statements $\phi_{i,j}$ exists via induction over the length of the trace execution.

We consider an interleaved trace of the two threads i and j that leads to a final variable assignment fulfilling the condition linking to statements $\phi_{i,j}$ backwards and show that for each postfix of the trace the existence implies the existence of the related satisfying assignment for the well formed formulas Φ'_i and Φ'_j of postfixes of the two threads i and j related to the postfix of the trace considered so far. This induction terminates as for any invalid safety property a finite trace invalidating it must exist.

Basis. For an empty statement for both thread any property that is not unsatisfiable can be fulfilled.

Induction Hypothesis. Let n be the length of trace execution. Then $\forall n, 1 \leq i \leq n, \phi_{i,j} \equiv T$

Induction Steps. The induction hypothesis holds for n^{th} length of trace execution and it contains all essential program constructors like assignment statement, important unary and binary operations, array expression, conditional statement, for loop etc. Here we show that the induction step holds for the assignment and the for-loop for brevity. Similar proofs hold for the rest of the program constructs.

Assignment. Let η_p and η_q be the the assignments statements of thread i and thread j respectively. The constructed well formed formulae Φ_i and Φ_j

has been generated by the range of the induction hypothesis, i.e., $\forall n, 1 \leq i, j \leq n$. For $(n+1)^{th}$ iteration, the well formed formulae must be constructed as the induction hypothesis holds.

For-Loop. Let η_p and η_q be the the assignments statements of therad i and thread j respectively. Let the loop has been unrolled n times and during unrolling the induction hypothesis holds. For unrolling once more times, the assertion becomes true.

For $(n+1)^{th}$ length the rest of program contracts are also available. Hence the induction hypothesis is true for $(n+1)^{th}$ length. $\square$

We now discuss first on the basis of this general results the termination and soundness of the data race detection algorithm.

5.3.2 Data Race: Soundness

Theorem 2. *If the function* `DataraceDetection` *(Algorithm 1) reports a data race for operation pair $\langle op_i^l, op_j^m \rangle$ where op_i^l be the operation i in l^{th} statement and op_j^m be the operation j in the m^{th} statement, then there is a data race between l^{th} and j^{th} statements.*

Proof. Let the step 10 of the function `DataraceDetection` (Algorithm 1) be true. However, the algorithm reports that there is no data race. Therefore, according to Theorem 1 the welformed formulae Φ_i and Φ_j are not formed for the threads i and j respectively. Therefore, step 9 is not satisfied by the condition and if step 9 is not satisfied, then the control does not go to step 10. Hence contradiction. $\square$

We now discuss the termination and soundness of the barrier divergence detection algorithm.

5.3.3 Barrier-divergence: Soundness

Theorem 3. *If the function* `BarrierDivergence` *reports a divergent barrier at statement s_i, then there must be a barrier call at s_i which is indeed divergent.*

Proof. Assume that the function `BarrierDivergence` reports no barrier divergence. Let us also assume that the statement s_b is a barrier statement. As s_b is a barrier statement, step 9 of the function `BarrierDivergence` must be satisfied. Therefore, according to Theorem 1 the welformed formulae Φ_i and Φ_j are not formed for the threads i and j respectively. Therefore, step 8 is not satisfied by the condition and if step 8 is not satisfied, then the control does not go to step 9. Hence contradiction. $\square$

Table 1: Results for Data race and Barrier divergence free detection times in seconds.

Benchmarks	LOC	#Kernels	#Conjuncts	DR	BD	DR Time (sec)	BD Time (sec)
CG	100	12	124	NO	NO	0.2314	0.1235
BCM	53	3	78	NO	NO	0.0031	0.0014
MIN_MAX	80	4	123	NO	NO	0.0120	0.0064
LUP	136	18	167	NO	NO	0.5195	0.3432
K-Means	142	34	231	NO	NO	0.8342	0.4123
HC	132	26	143	NO	NO	0.4167	0.2437
DCT	30	2	53	NO	NO	0.0075	0.0042
LRU	190	23	272	NO	NO	0.9198	0.6342
FFT	1002	47	1845	NO	NO	1.83455	0.9943
UA	1234	172	1324	NO	NO	2.3412	2.0034

5.4 Complexity Analysis

We discuss the complexity of the verification algorithms for data race detection and barrier divergence, Algorithm 1 and Algorithm 2 respectively in a rigorous way.

Algorithm 1. Step 1 takes $O(1)$ time. Step 2 computes pair of conflict statements and it takes $O(n^2)$ time. Step 3 and 4 take $O(1)$ times. Step 6 computes pairs of conflicting operators and if the number of operators are $|F|$, then it takes $O(|F|^2)$ time. SMT solver is called in Step 8 for generating SMT equation which takes $O(2^{|F|})$ where F is the length of the constraint. Only addition operation takes place in the if-block (Steps 9-11) and takes $O(1)$ time as we maintained a value indexed data structure. In Step 12, only deletion operation takes place and takes $O(1)$ time. The loop in step 7 iterates $|F|$ times. Hence overall complexity is $O(|F|.2^{|F|})$ times. Similarly, in step 5, the loop iterates n times; the total complexity is $O(n.|F|.2^{|F|}) \simeq O(2^{|F|})$.

Algorithm 2. Step 1 takes $O(1)$ time. Step 2 computes all the barriers for a program and it takes $O(n)$ time. Step 3 and 4 take $O(1)$ times. Step 5 computes pairs of barrier and it takes $O(n^2)$ time. SMT solver is called in Step 7 for generating SMT equation which takes $O(2^{|F|})$ where F is the length of the constraint. Only addition operation takes place in the if-block (Steps 8-10) and takes $O(1)$ time as we maintained a value indexed data structure. In Step 11, only deletion operation takes place and takes $O(1)$ time. The loop in step 6 iterates n times. Hence the overall complexity is $O(n.2^{|F|}) \simeq O(2^{|F|})$ times.

6 EXPERIMENTAL RESULTS

The verification algorithm is implemented in `Python` and tested on several parallel benchmarks on a 2.0 GHz Intel(R) Core(TM)2 Duo CPU machine (using only a single core). The set of benchmarks is available in (Bondhugula et al., 2008; Che et al., 2009). The steps for carrying out the experimentation are outlined below.

1. **Preparation of the Benchmarks Suite.** The source code and the documentation related to the list of source programs are given in (Bondhugula et al., 2008; Che et al., 2009). It is to be noted that some of these examples such as K-Means, HC, DCT, LRU, UA, and FFT are widely used in various application domains such as bioinformatics, image processing, computational fluid dynamics (CFD), etc.
2. **Program to WFF.** Each of the above programs is then translated into the well-formed formulae by the translation module of the tool as described in Section 4. Next, the set of well-formed formulae is used by the verification modules to detect data race and barrier divergence conditions.
3. **Reporting of Results.** For each test case, we observed the numbers of kernels, the number of conjuncts in SMT equations and the number of data race and barrier divergence are encountered. For small examples like DCT and MINANDMAX, the data race and barrier divergence are manually verified for correctness.

6.1 Analysis

Table 1 represents the description of the parallel benchmarks concerning the number of lines of code (LOC), number of kernels and number of conjuncts. It is to be noted that the number of conjuncts is always greater than the number of lines of codes because there exists at least one conjunct in each line of code and at least one additional conjunct should be present between any two lines of code. Table 1 also depicts that there is no data race and barrier divergence for each benchmark. Last two columns of Table 1 depicts the data race and barrier divergence detection time. In all the cases, it is also worth noting that detection of barrier divergence time is smaller than the data race detection. If barrier divergence occurs, then there is a possibility for data race. However, the reverse is not true.

Finally, we take some applications from the benchmark suites and manually inject some errors at the code level. The objective of this line of experimentation is to check the efficacy of the verification module in detecting data race and barrier divergence respectively. We have introduced the following types of error.

Type 1. Removal of barrier where read after write operation occurs between two threads; this has been injected in the `HC` and `DCT` benchmarks.

Type 2. Removal of the barrier where a write after a read operation occurs between two threads; this has been injected in the `LUP` benchmark.

Table 2: Results for Data race and Barrier divergence detection times.

Error	Benchmarks	#DR	#BD	DR Time (sec)	BD Time (sec)
Type 1	HC	6	5	0.8314	0.5235
	DCT	2	2	0.0234	0.0123
Type 2	LUP	5	3	0.6298	0.4231
Type 3	HC	6	5	0.7217	0.6355
Type 4	MINANDMAX	3	1	0.1237	0.0934
	UA	5	4	0.3256	0.2651

Type 3. Removal of the barrier where write after write operation occurs between two threads; this has been injected in the `HC` benchmark.

Type 4. data-locality error which introduce false data-locality in the conditional branch in `MINANDMAX` and `UA` benchmarks.

Table 3: Characteristics of counter examples.

Error	Benchmarks	Counter Model size	
		# formulae	# conjuncts
Type 1	HC	12	45
	DCT	7	24
Type 2	LUP	5	16
Type 3	HC	12	41
Type 4	MINANDMAX	3	9
	UA	24	89

Table 3 only considered the part where the errors have been injected rather than the entire code. For this reason, the number of formulae or the number of conjuncts vary from Table 2.

6.2 Analysis of Erroneous Benchmarks

Table 2 depicts the descriptions of the errors introduced in the benchmarks, the number of data races and barrier divergences and the execution times taken by the data race detection module and by the barrier divergence module; (in each cases, the data race and barrier divergence have been detected by the modules successfully;) The last two columns of the Table 2 record the positive data race and barrier divergence detection time. It is to be noted that in all cases, the actual data race and barrier divergence detection time is higher than the data race free and barrier divergence free detection time. If the verification module detects that there is a data race or barrier divergence, the tool constructs the corresponding counter models and gives the counter model as the output. For this reason, real data race and barrier divergence detection time are much higher. Table 3 depicts the size of the counter model concerning the number of formulae and number of conjuncts.

6.3 Analysis of Comparative Study

We have compared the performance of our tool with the online version of PUG as shown in Table 4. The noteworthy fact is that our tool can identify a set of barrier divergence and data races that span across multiple barriers in a single iteration. Since PUG stops after detecting barrier divergences and data races present within the first barrier it encounters, it requires multiple runs (after correcting the already encountered problems) to discover additional barriers spanning across multiple barriers. From the developer productivity point of view, our approach certainly helps to repair the program faster.

Table 4: Comparative results.

Error	Benchmarks	# Barrier Divergence	# Data Races	#Iterations	
				PUG	Our Tool
Type 1	HC	6	5	2	1
	DCT	2	2	1	1
Type 2	LUP	5	3	2	1
Type 3	HC	6	5	3	1
Type 4	MIN_MAX	3	1	2	1
	UA	5	4	3	1

7 CONCLUSION

In this paper, we have proposed an approach for static verification of GPU kernels programmed using CUDA. Our approach verifies whether the program has the data race or has barrier divergence issues. Our approach can identify all the violations of these across the barriers for which we can generate witnesses in the code. We have elaborated the implementation of our approach and provided a formal proof of the soundness of the algorithm. Thus, if our tool reports a data race or a barrier divergence, these conditions do exist in the code. Like other reported approaches, our tool does not have a completeness property, i.e., it cannot report all possible data-races or barrier divergences. The experimental results indicate that the proposed tool has a minimal performance impact.

Producing a list of conflicts in one go has benefits for verification tool developers, where the tool can take advantage of the list of conflicts and barrier divergences and suggest placement of barriers in the code to remove the conflicts. This assistance mechanism can help programmers to implement conflict free CUDA programs more efficiently.

We are working to include verification of additional properties such as shared memory bank conflicts and global coalesced memory accesses as they can significantly affect the performance of the kernel and thus impact the observed speedup. We also plan to build this verifier as an integral part of the design process so that the designers can check for these safety properties as they develop the code for GPGPU.

ACKNOWLEDGEMENTS

This work is financially supported by Science and Engineering Research Board, Govt. of India funding (SB/S3/EECE/0170/2014).

REFERENCES

(2017). Cuda c programming guide. Technical Report PG-02829-001_v8.0, NVIDIA Corporation.

Bandyopadhyay, S., Sarkar, D., and Mandal, C. (2018). Equivalence checking of petri net models of programs using static and dynamic cut-points. *Acta Informatica*.

Bandyopadhyay, S., Sarkar, D., Mandal, C. A., Banerjee, K., and Duddu, K. R. (2016). A path construction algorithm for translation validation using PRES+ models. *Parallel Processing Letters*, 26(2):1–25.

Bandyopadhyay, S., Sarkar, S., Sarkar, D., and Mandal, C. A. (2017). Samatulyata: An efficient path based equivalence checking tool. In *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings*, pages 109–116.

Bardsley, E. and Donaldson, A. F. (2014). *Warps and Atomics: Beyond Barrier Synchronization in the Verification of GPU Kernels*, pages 230–245. Springer International Publishing, Cham.

Betts, A., Chong, N., Donaldson, A. F., Ketema, J., Qadeer, S., Thomson, P., and Wickerson, J. (2015). The design and implementation of a verification technique for gpu kernels. *ACM Trans. Program. Lang. Syst.*, 37(3):10:1–10:49.

Blom, S. and Huisman, M. (2014). *The VerCors Tool for Verification of Concurrent Programs*, pages 127–131. Springer International Publishing.

Bondhugula, U., Hartono, A., Ramanujam, J., and Sadayappan, P. (2008). Pluto: A practical and fully automatic polyhedral program optimization system. In *PLDI 08*.

Boyer, M., Skadron, K., and Weimer, W. (2008). Automated Dynamic Analysis of CUDA Programs. In *Third Workshop on Software Tools for MultiCore Systems*.

Che, S., Boyer, M., et al. (2009). Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IEEE Intl. Symp. on Workload Characterization (IISWC)*.

Chiang, W.-F., Gopalakrishnan, G., Li, G., and Rakamarić, Z. (2013). *Formal Analysis of GPU Programs with Atomics via Conflict-Directed Delay-Bounding*, pages 213–228. Springer Berlin Heidelberg, Berlin, Heidelberg.

Chong, N., Donaldson, A. F., Kelly, P. H., Ketema, J., and Qadeer, S. (2013). Barrier invariants: A shared state abstraction for the analysis of data-dependent gpu kernels. OOPSLA '13, pages 605–622, New York, NY, USA. ACM.

Collingbourne, P., Cadar, C., and Kelly, P. H. J. (2012). Symbolic testing of opencl code. In *Proceedings of the 7th International Haifa Verification Conference on Hardware and Software: Verification and Testing*, HVC'11, pages 203–218, Berlin, Heidelberg. Springer-Verlag.

Collingbourne, P., Donaldson, A. F., Ketema, J., and Qadeer, S. (2013). *Interleaving and Lock-Step Semantics for Analysis and Verification of GPU Kernels*, pages 270–289. Springer Berlin Heidelberg, Berlin, Heidelberg.

Darte, A. and Schreiber, R. (2005). A linear-time algorithm for optimal barrier placement. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 26–35, New York, NY, USA. ACM.

Dhok, M., Mudduluru, R., and Ramanathan, M. K. (2015). Pegasus: Automatic barrier inference for stable multithreaded systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 153–164, New York, NY, USA. ACM.

Ernstsson, A., Li, L., and Kessler, C. (2017). SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. *International Journal of Parallel Programming*.

Hathhorn, C., Becchi, M., Harrison, W. L., and Procter, A. M. (2012). Formal semantics of heterogeneous CUDA-C: A modular approach with applications. In *Proceedings Seventh Conference on Systems Software Verification, SSV 2012, Sydney, Australia, 28-30 November 2012.*, pages 115–124.

Huisman, M. and Mihelčić, M. (2013). Specification and verification of GPGPU programs using permission-based separation logic. In *8th Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE 2013)*.

Kirk, D. and Hwu, W.-M. W. (2016). *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 3 edition.

Lamport, L. (1977). Proving the correctness of multprocess programs. *IEEE Trans. Software Engineering*, 3(2).

Lattner, C. and Adve, V. (2004). LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. Intl. Symp. on Code generation and optimization*.

Leung, A., Gupta, M., Agarwal, Y., Gupta, R., Jhala, R., and Lerner, S. (2012). Verifying gpu kernels by test amplification. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 383–394.

Li, G. (2010). *Formal verification of programs and their transformations*. PhD thesis.

Li, G. and Gopalakrishnan, G. (2010). Scalable smt-based verification of gpu kernel functions. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*, pages 187–196.

Li, G. and Gopalakrishnan, G. (2012). Parameterized verification of gpu kernel programs. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IPDPSW '12, pages 2450–2459.

Li, G., Li, P., Sawaya, G., Gopalakrishnan, G., Ghosh, I., and Rajan, S. P. (2012a). GKLEE: Concolic Verification and Test Generation for GPUs. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 215–224.

Li, P., Li, G., and Gopalakrishnan, G. (2012b). Parametric flows: Automated behavior equivalencing for symbolic analysis of races in cuda programs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 29:1–29:10.

Lin, Z., Gao, X., Wan, H., and Jiang, B. (2015). *GLES: A Practical GPGPU Optimizing Compiler Using Data Sharing and Thread Coarsening*, pages 36–50. Springer, Cham.

Lv, J., Li, G., Humphrey, A., and Gopalakrishnan, G. (2011). Performance Degradation Analysis of GPU Kernels. In *CAV EC^2*.

Manna, Z. (1974). *Mathematical Theory of Computation*. McGraw-Hill Kogakusha, Tokyo.

McCarthy, J. (1962). Towards a Mathematical Science of Computation. In *IFIP Congress*, pages 21–28.

Nickolls, J. and Dally, W. J. (2010). The GPU computing era. *IEEE Micro*, 30(2).

Oboyle, M., Kervella, L., and Bodin, F. (1995). Synchronization minimization in a spmd execution model. *Journal of Parallel and Distributed Computing*, 29(2):196 – 210.

Said, M., Wang, C., Yang, Z., and Sakallah, K. (2011). Generating data race witnesses by an smt-based analysis. In *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, pages 313–327. Springer-Verlag.

Stöhr, E. A. and O'Boyle, M. F. P. (1997). A graph based approach to barrier synchronisation minimisation. In *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, pages 156–163.

Zheng, M., Ravi, V. T., Qin, F., and Agrawal, G. (2011). GRace: A Low-overhead Mechanism for Detecting Data Races in GPU Programs. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 135–146.

Zheng, M., Ravi, V. T., Qin, F., and Agrawal, G. (2014). Gmrace: Detecting data races in gpu programs via a low-overhead scheme. *IEEE Transactions on Parallel and Distributed Systems*, 25(1):104–115.