Bounded Exhaustive Test-Input Generation on GPUs

AHMET CELIK, The University of Texas at Austin, USA
SREEPATHI PAI, The University of Texas at Austin, USA
SARFRAZ KHURSHID, The University of Texas at Austin, USA
MILOS GLIGORIC, The University of Texas at Austin, USA

Bounded exhaustive testing is an effective methodology for detecting bugs in a wide range of applications. A well-known approach for bounded exhaustive testing is Korat which generates all test inputs up to a given small size based on a formal specification that characterizes properties of desired test inputs. This specification is written as an executable predicate and Korat executes the predicate on candidate inputs to implement a backtracking search based on pruning to systematically explore the space of all possible inputs and generate only those that satisfy the specification.

This paper presents a novel approach for speeding up test generation for bounded exhaustive testing using Korat. The novelty of our approach is two-fold. One, we introduce a new approach for writing the specification predicate based on an abstract representation of candidate inputs, so that the predicate executes directly on these abstract structures and each execution has a lower cost. Two, we use the abstract representation as the basis to define the first technique for utilizing GPUs for systematic test generation using executable predicates. Moreover, we present a suite of optimizations that are necessary to enable effective utilization of the computational resources offered by modern GPUs. We use our prototype tool to experimentally evaluate our approach using a suite of 7 data structures that were used in prior studies on bounded exhaustive testing. Our results show that our abstract representation can speed up test generation by 5.68× on a standard CPU, while execution on a GPU speeds up the generation, on average, by 17.46×.

CCS Concepts: • **Software and its engineering → Software testing and debugging**;

Additional Key Words and Phrases: Bounded exhaustive testing, graphics processing units, intKorat

ACM Reference Format:
Ahmet Celik, Sreepathi Pai, Sarfraz Khurshid, and Milos Gligoric. 2017. Bounded Exhaustive Test-Input Generation on GPUs. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 94 (October 2017), 25 pages. https://doi.org/10.1145/3133918

1 INTRODUCTION

Software testing is the most common approach in industry to check correctness of software. However, writing tests manually is tiresome and time consuming.

Researchers (and practitioners) have developed a number of automated test generation techniques [Bertolino 2007; Boyapati et al. 2002; Duncan and Hutchison 1981; Pacheco et al. 2007; Tai and Lei 2002; Tonella 2004] to remove the burden from developers. These automated techniques

Author's Addresses: Ahmet Celik, The University of Texas at Austin, Austin, TX 78712; email: ahmetcelik@utexas.edu; Sreepathi Pai, (Current Address) University of Rochester, Rochester, NY 14627; email: sree@cs.rochester.edu, Sarfraz Khurshid, The University of Texas at Austin, Austin, TX 78712; email: khurshid@utexas.edu; Milos Gligoric, The University of Texas at Austin, Austin, TX 78712; email: gligoric@utexas.edu.

2475-1421/2017/10-ART94
https://doi.org/10.1145/3133918

have been shown effective for finding bugs in a wide range of applications [Daniel et al. 2007; Gligoric et al. 2010; Yang et al. 2011]. One approach to automated test generation is *bounded exhaustive testing* [Marinov and Khurshid 2001] (BET), which generates all test cases up to the given bound. Many BET techniques have been proposed over the last decade [Boyapati et al. 2002; Gligoric et al. 2010; Korat Home Page 2017; Kuraj et al. 2015; Misailovic et al. 2007; Ponzio et al. 2016; Rosner et al. 2014; Siddiqui and Khurshid 2009; Sullivan et al. 2004].

One of the most popular BET techniques is Korat [Boyapati et al. 2002]. It generates all test cases (up to a given size) based on a formal specification that characterizes properties of desired inputs. Korat allows users to write the specification as an executable predicate in the same language as the program under test. Specifically, the user provides: (1) the bounds that define the space of candidate inputs Korat should *explore*, e.g., maximum size of an object graph, and (2) the predicate, traditionally termed `repOK` [Liskov and Guttag 2000], which returns true if and only if its input satisfies the desired properties. Korat explores the bounded space by creating candidate inputs within the bounds and executing the `repOK` predicate on each candidate. Korat uses the predicate's executions on candidate inputs to perform a backtracking search based on *pruning* and systematically explores the space of all possible inputs to generate only those that satisfy the specification. The generated inputs are then used for test execution.

Despite Korat's pruning, unfortunately, the search space quickly becomes intractable even for small bounds which results in slow *test generation* and *test execution* [Boyapati et al. 2002; Misailovic et al. 2007; Siddiqui and Khurshid 2009]. To improve the performance of Korat, researchers and practitioners previously considered parallel algorithms. Misailovic et al. [2007] developed *Parallel Korat*, a set of algorithms for generation and execution on map-reduce model. More recently, Siddiqui and Khurshid [2009] proposed *PKorat*, a scalable algorithm that significantly improves the scalability and scope of parallel *test generation*.

Although PKorat sped up test generation process by 7.05×, on average, it has two key limitations. First, it requires access to a large cluster of machines, which may not be readily accessible to practitioners and researchers worldwide. Second, as the original Korat, it is not suitable for execution on Graphics Processing Units (GPUs), because it uses dynamic memory allocation, libraries, and operators that are inefficient on GPUs. Our initial effort to migrate the Korat technique to GPUs led to (insignificant) 1.19× speedup on a GeForce GTX 1080 and 0.43× speedup (i.e., slowdown) on a Tesla K80. Furthermore, Korat and PKorat assume that virtual memory is available during test generation, which is not the case for most GPUs. Therefore, we could not even run the test generation for larger sizes on GPUs.

This paper presents IntKorat, a novel approach for speeding up test generation for bounded exhaustive testing. The novelty of our approach is two-fold. One, we introduce a new approach for writing the *specification predicate based on an abstract representation of candidate inputs*, so that the predicate executes directly on these abstract structures and each execution has a lower cost. Predicates for IntKorat use candidates that are encoded as arrays of integers (rather than as object graphs in memory, which was the traditional approach). Since during test generation the predicate is typically executed a large number of times, the abstract representation significantly speeds up the overall exploration. A minor contribution of this paper is an *automated migration procedure* from the existing specifications with object graphs to the new specifications with int arrays. Two, we use the abstract representation as the basis to define *the first technique for utilizing GPUs for systematic test generation* using executable predicates. Our abstract representation naturally lends an effective input encoding for GPUs since any code that extensively accesses objects is unlikely to scale when run on a GPU. Furthermore, to overcome the lack of virtual memory on GPUs, our implementation uses a worklist-based algorithm, and to overcome inefficient computations, we introduce a suite of

GPU-specific optimizations that enable effective utilization of the computational resources offered by modern GPUs.

We implemented IntKorat for three platforms: Java (novel specification only), C (novel specification only), and NVIDIA's CUDA for GPUs. We evaluated our implementation for each platform and compared with the original Korat implementation that invokes `repOK` on object graphs. We find that our technique for writing predicates that operate on abstract structures can frequently speed up test generation process. Further, we find that our implementation for CUDA speeds up test generation process by 17.46× (on average). Thus, our contributions enable test generation of larger test inputs in the given time, and prior research has shown that larger sizes can find unique bugs that cannot be exposed otherwise [Gligoric et al. 2010; Nokhbeh Zaeem and Khurshid 2012].

The key contributions of this paper include:

- Novel approach, dubbed IntKorat, for writing imperative predicates that operate on abstract structures based on arrays of integers rather than on in-memory object graphs. IntKorat speeds up test generation on CPUs and is a natural and effective input encoding for GPUs.
- The first worklist algorithm, to the best of our knowledge, that works around the lack of virtual memory on GPUs and allows them to be used for systematic test generation of larger test inputs.
- A set of GPU-specific optimizations that enables efficient test generation by utilizing the computational resources offered by modern GPUs.
- Implementation of IntKorat for three platforms: Java (includes novel encoding only), C (includes novel encoding only), and NVIDIA's CUDA (includes novel encoding, the worklist algorithm, and GPU-specific optimizations).
- Evaluation of our implementations on 7 data structures, which were commonly used in prior work on test generation. Our results show that our technique based on abstract structures can lead to speedup of test generation. More importantly, our results show that the implementation for CUDA speeds up the generation, on average, by 17.46×.

2 OVERVIEW

This section describes Korat through an example and provides a brief overview of GPGPU programming model.

2.1 Korat and PKorat

We illustrate the original Korat algorithm using Binary Tree data structure as an example. Figure 1a shows a code snippet in Java that defines two classes: `BT` and `Node`. `BT` describes an instance of a binary tree; each instance keeps the size of the tree (`size`) and a pointer to the root of the tree (`root`). Each node in the tree, described with the class `Node`, contains an element (which has the integer type in our example) and pointers to left and right children.

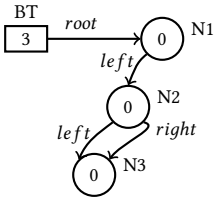
Valid instances of `BT` should be acyclic and have the value in the `size` field that matches the number of nodes that are reachable from the `root`. Figure 2a shows an imperative predicate (`repOK`) that returns `true` if a given instance of `BT` is *valid* and `false` otherwise. Predicates for Korat are written in Java.

Additionally, a user of Korat has to specify the bounds, i.e., possible values for each field of each class. This method is frequently called finitization, and we show an example in Figure 3.

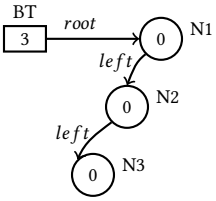
Korat's generation engine takes as input: (1) classes that define the structure, (2) the finitization, and (3) the predicate. Korat outputs all instances that are valid, i.e., instances for which `repOK` returns `true`. Internally, Korat represents each instance with an array of integers, called a *candidate vector*. Figures 1d and 1e show two candidate vectors and Figures 1b and 1c show the corresponding

```
1  class BT {
2    Node root;
3    int size;
4  }
5
6  class Node {
7    int element;
8    Node left;
9    Node right;
10 }
```

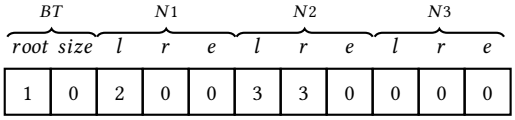
(a) Class definition for our running example – Binary Tree



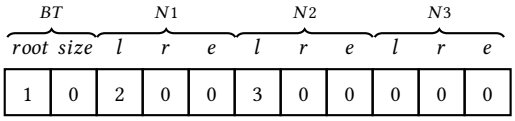
(b) An example object graph for an invalid binary tree instance



(c) An example object graph for a valid binary tree instance



(d) An example candidate vector that encodes the invalid instance from Figure 1b



(e) An example candidate vector that encodes the valid instance from Figure 1c

Fig. 1. Binary Tree example

in-memory object graphs. Prior to invoking the predicate, Korat converts each candidate vector to an object graph.

Naive Korat exploration enumerates all candidate vectors up to the bounds specified in the finitization. Even for small sizes this quickly becomes intractable. For example, naive exploration enumerates 16,384 candidates to generate 30 valid instances of BT of size 3. Interestingly, naive exploration is embarrassingly parallel.

To reduce the search space, Korat keeps an ordered list of *accessed fields* during the execution of the predicate [Boyapati et al. 2002]. We refer to the exploration with the accessed fields as *accessed-field optimized exploration*. Korat records each field only once and only the first access to the field. If a predicate evaluates to `false`, Korat skips all candidate vectors that would differ only in the values of fields that are never accessed; fields that are not accessed cannot change the outcome of the predicate. In our example in Figure 1d, predicate evaluates to `false` and the accessed fields include 0, 2, 3, 5, 6 (which corresponds to `BT.root`, `N1.left`, `N1.right`, `N2.left`, and `N2.right`). Therefore, changing the value of the 8th element (i.e., `N3.left`) would have no impact on the evaluation of the predicate.

Although accessed-field optimized exploration substantially reduces the number of candidates to explore, e.g., only 63 candidates for BT of size 3, it makes the algorithm mostly sequential [Misailovic et al. 2007; Siddiqui and Khurshid 2009].

Recent work [Siddiqui and Khurshid 2009] introduced PKorat that uses accessed fields to identify candidates that eventually need to be explored and can be explored in parallel. However, PKorat

```
1  boolean repOK() {
2    if (root == null)
3      return size == 0;
4    // checks that tree has no cycle
5    Set visited = new HashSet();
6    visited.add(root);
7    LinkedList workList = new LinkedList();
8    workList.add(root);
9    while (!workList.isEmpty()) {
10     Node current = (Node) workList.rmFirst();
11     if (current.left != null) {
12       if (!visited.add(current.left))
13         return false;
14       workList.add(current.left);
15     }
16     if (current.right != null) {
17       if (!visited.add(current.right))
18         return false;
19       workList.add(current.right);
20     }
21   }
22   // checks that size is consistent
23   return (visited.size() == size);
24 }
```

(a) Predicate for original Korat

```
1  boolean repOK() {
2    if ((getValue("Node",
3                  getIndex("BT", "root", 1))) == 0)
4      return (getValue("Size",
5                       getIndex("BT", "size", 1))) == 0;
6    Set visited = new HashSet();
7    visited.add(getValue("Node",
8                         getIndex("BT", "root", 1)));
9    LinkedList worklist = new LinkedList();
10   worklist.add(getValue("Node",
11                        getIndex("BT", "root", 1)));
12   while (!(worklist.isEmpty())) {
13     Integer current = (Integer) worklist.rmFirst();
14     if ((getValue("Node",
15             getIndex("Node", "left", current))) != 0) {
16       if (!(visited.add(getValue("Node",
17               getIndex("Node", "left", current)))))
18         return false;
19       worklist.add(getValue("Node",
20                     getIndex("Node", "left", current)));
21     }
22     if ((getValue("Node",
23           getIndex("Node", "right", current))) != 0) {
24       if (!(visited.add(getValue("Node",
25                     getIndex("Node", "right", current)))))
26         return false;
27       worklist.add(getValue("Node",
28                getIndex("Node", "right", current)));
29     }
30   }
31   return (getValue("Size",
32         getIndex("BT", "size", 1))) == (visited.size());
33 }
```

(b) Predicate for intKorat

Fig. 2. (left) The predicate for the original Korat checks a candidate encoded as an object graph, and (right) the predicate for intKorat checks a candidate encoded as an array of integer values. We defined a procedure for automated migration to the new encoding

evaluates predicates on in-memory object graphs, same as the original Korat, which introduces substantial overhead if executed on a GPU. Additionally, PKorat assumes that unlimited/virtual memory is available, which is not the case for most GPUs.

2.2 GPGPU Programming Model

Graphics Processing Units (GPUs) were originally designed to accelerate graphics computations, for example in 3D games. Compared to a CPU, GPUs support higher levels of parallelism (thousands of concurrent threads) and much higher memory bandwidth (hundreds of GB/s). Since most GPUs today are architected as a large number of general-purpose cores with only a few graphics-specific hardware blocks, they can also run a wide variety of general-purpose code. In particular, they perform very well on regular, data-parallel computations (e.g. dense matrix multiplication) where

```
1  IFinitization finBinaryTree(int numOfNodes, int size) {
2      IFinitization f = FinitizationFactory.create(BinaryTree.class);
3      IObjSet nodes = f.createObjSet(Node.class, numOfNodes);
4      nodes.setNullAllowed(true);
5      IIntSet sizes = f.createIntSet(size, size);
6      f.set("root", nodes);
7      f.set("size", sizes);
8      f.set("Node.left", nodes);
9      f.set("Node.right", nodes);
10     return f;
11 }
```

Fig. 3. Finitization to specify the bounds on instances to be generated

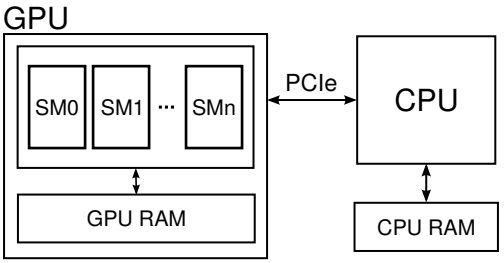


Fig. 4. Organization of an abstract GPU-equipped system

they are often an order of magnitude faster than CPUs [Lee et al. 2010]. As a result, GPUs are now a mainstay of the high-performance computing (HPC) community, contributing over 10% of the performance of the Top 500 supercomputers [Strohmaier et al. 2016].

Figure 4 presents an abstract view of a system equipped with a GPU. A GPU contains a number of cores called streaming multiprocessors (SMs). Each SM can execute thousands of threads. A GPU also contains its own memory which is directly addressable from the SMs. This GPU RAM uses a wider datapath, a different memory technology than CPUs (GDDR5X on the primary GPU we use) and operates at a high clock allowing the SMs to access data at hundreds of gigabytes per second. The CPU and GPU are connected via a PCIe link in the systems used in this paper.

To benefit from GPUs, programmers must rewrite their code to target the GPU. Originally, the only options were shader languages (used by graphics programs), but NVIDIA's CUDA pioneered the use of a general-purpose language for GPGPU. GPUs may also be programmed through the use of OpenCL, however OpenCL is poorly supported (e.g., lack of C++ constructs) on NVIDIA GPUs that we use in this work. Although our approach is conceptually language-agnostic, our implementation targets NVIDIA CUDA; we discuss limitations of OpenCL in detail in Section 6.

The CUDA programming language is a dialect of C++. CUDA source code consists of host code (executed by the CPU) and device code (executed by the GPU). Device code consists of *kernel* functions which are delineated by the `__global__` function qualifier. Each kernel is written in the single-program multiple data (SPMD) model similar to MPI programs. The CUDA compiler, `nvcc`, extracts these kernels from the source code and compiles them, leaving behind only the CPU code and stubs for calling GPU kernels. This CPU code is then compiled by the system C++ compiler.

Programmers execute GPU kernels by *launching* them from the CPU code. This is similar to a function call, except that the kernel will execute on the GPU and that programmers must specify a launch configuration for the kernel. The launch configuration, known as a grid, specifies the

number of threads that will be created for the kernel. It is a hierarchical structure consisting of equally-sized thread blocks which contain the threads. CUDA uses a hierarchical grid instead of a flat space of threads for portability across multiple GPUs. Additionally, only threads within the same thread block can reliably communicate and synchronize with each other.

Kernels are always launched asynchronously – the CPU does not wait for the kernel to finish. This allows the CPU and GPU to work concurrently. The CPU can use CUDA synchronization primitives to wait for a kernel when required.

Modern GPU kernels can directly access data on other GPUs and the CPU through load/store instructions, however this is slower than accessing GPU RAM. To obtain the best performance, the programmer must transfer to GPU memory all data accessed by the kernel using explicit memory copies. Results must be copied back to the CPU after a kernel finishes. Most GPUs have limited memory (12–24GB) and only the latest NVIDIA Pascal-based GPUs support virtual memory [NVIDIA 2016]. As we will show later, using this GPU virtual memory functionality results in slower execution, so it is still necessary to limit memory usage to the size of GPU memory.

intKorat benefits substantially from the large degree of parallelism and the large memory bandwidth offered by GPUs. There are two main complications in implementing intKorat for GPUs. First, when using accessed-field optimized exploration, the optimization causes each thread to process a different number of candidate vectors leading to work imbalance among the threads. The effect of this load imbalance can be reduced by dynamically reassigning work to other threads. Thus, test generation can be viewed as *irregular* producer/consumer parallelism, where a thread works on a candidate vector and places all new candidate vectors in a queue so they will be processed by other threads. Second, since the number of new candidate vectors is not known in advance and GPU memory size is limited, we must use sophisticated queue management techniques during execution to prevent queue overflows, especially when the number of test candidates is very large. Alternatively, we could use newly-available GPU virtual memory mechanisms, but our experiments show that these significantly slow down execution.

3 TECHNIQUE

We describe our technique in two parts. First, we describe the technical contributions necessary to *enable* the bounded exhaustive testing on GPUs: GPU-friendly encoding of predicates (Section 3.1) and the worklist algorithm (Section 3.2). Second, we describe several code transformations to *optimize* the test generation process and make the algorithm efficient (sections 3.3 and 3.4).

3.1 Encoding

Our GPU-friendly encoding is the first step necessary to enable the bounded exhaustive testing on GPUs. The goal is to remove code manipulations of object graphs and instead use manipulations of integer arrays that is an ideal fit for GPUs. Figures 10a and 10b (in the Appendix) illustrate the encoding step.

The objective of our encoding is to transform an object graph into a *ragged candidate matrix* which contains only integers. Then, we must rewrite the `repOK` function to operate on this ragged candidate matrix instead of the original object graph. Similarly, all functions (recursively) reachable from `repOK` must be rewritten to use the ragged candidate matrix. Before we describe our rewrite procedure, we show how object graphs can be encoded as integers.

3.1.1 Preliminaries. We define a Type as a structure similar to a class in Java. We use Θ to denote a set of all types used in a finitization. Further, we write $\breve{\Theta}$ (s.t., $\breve{\Theta} \subseteq \Theta$) and $\widehat{\Theta}$ (s.t., $\widehat{\Theta} \subseteq \Theta$) to denote a set of composite and primitive types, respectively. The domain of a type θ (s.t., $\theta \in \Theta$), written as $\Delta(\theta)$, is a finite set of values.

Table 1. Types and Their Domains for Our Running Example

Kind	Type	Domain	Function
Composite	BT	$[1, 1]$	N/A
Composite	Node	$[0, 3]$	N/A
Primitive	Size	$[3, 3]$	$f : \{3\} \rightarrow \{3\}, f(x) = x$
Primitive	Element	$[0, 0]$	$g : \{0\} \rightarrow \{0\}, g(x) = x$

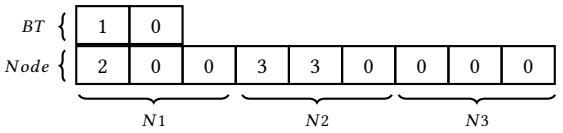


Fig. 5. Ragged candidate matrix for our running example

$$\text{RDomainValue}\quad \frac{\Gamma \vdash x \in \Delta(\theta), \theta \in \Theta}{R(x) \rightarrow x}$$

$$\text{RImplFieldRef}\quad \frac{\Gamma \vdash x : \theta, (x, \theta) \in \breve{\theta}}{R(x) \rightarrow getValue(\theta, getIndex(\breve{\theta}, \text{“}x\text{”}, 1))}$$

$$\text{RFieldRef}\quad \frac{\Gamma \vdash x.y : \theta, x : \breve{\theta}}{R(x.y) \rightarrow getValue(\theta, getIndex(\breve{\theta}, \text{“}y\text{”}, R(x)))}$$

Fig. 6. Rewrite rules for migrating variable and field accesses in `repOK` and other functions

A composite type $\breve{\theta} \in \breve{\Theta}$ is an ordered collection of fields $< f_1, f_2, ..., f_n >$. Each field is a pair of a name and a type, i.e., $f_1 = (fieldName_1, \theta_1), f_2 = (fieldName_2, \theta_2), ..., f_n = (fieldName_n, \theta_n)$ where $\theta_1, \theta_2, ..., \theta_n \in \Theta$. A name of a field is a sequence of characters unique within the declaring type. Function $\text{getType}(\breve{\theta}, fieldName)$ returns the type of the field declared in $\breve{\theta}$. The domain of a composite type $\Delta(\breve{\theta})$ is a sequence of integers starting either from 0 or 1. Each value in the domain, which we call instanceId, corresponds to a unique object instance created in the original finitization. The value 0 belongs to the domain only if the `null` value is permitted.

A primitive type $\widehat{\theta} \in \widehat{\Theta}$ defines a set of values. The domain of a primitive type $\Delta(\widehat{\theta})$ is an interval $[min\ ..\ max]$. Additionally, we associate a bijective mapping function to each domain.

Running example: Table 1 shows Θ for our running example. There are two composite ($\breve{\theta} = \{\texttt{BT}, \texttt{Node}\}$) and two primitive ($\widehat{\theta} = \{\texttt{Size}, \texttt{Element}\}$) types. $\Delta(\texttt{BT}) = [1\ ..\ 1]$, as we only have a single `BT` instance and the `null` value is not permitted. $\Delta(\texttt{Node}) = [0\ ..\ 3]$, as we created three unique instances of `Node` and the `null` value is permitted. In addition, we have two primitive types for two fields: `size` (defined in `BT`) and `elements` (defined in `Node`); each of these fields has a single value. Finally, we need to associate a mapping function to each primitive type; we use the identity function as the mapping function. (In case of discrete values, e.g., Red and Black, we would define the mapping as $g : \{0, 1\} \rightarrow \{Black, Red\}$ and $g = \{(0, Black), (1, Red)\}$.)

We define a *target type* $\tau \in \breve{\Theta}$ as a composite type for which to generate instances. τ is `BT` for our running example. We represent an instance of the τ with a ragged candidate matrix (*matrix*

for short). Each row of a matrix corresponds to a single composite type $\breve{\theta} \in \breve{\Theta}$. To simplify our encoding, we always allocate the first row for τ. We compute the number of cells for the row of type $\breve{\theta}$ as: "number of fields" $\times$ "max number of instances", such that the "max number of instances" is the max value in $\Delta(\breve{\theta})$. Each cell of the matrix contains an index to be used to get the value from the domain $\Delta(\theta)$. For example, cell with value 1 "points to" the second value in the $\Delta(\theta)$. Figure 5 shows the matrix for our running example.

We define a function getIndex($\breve{\theta}$, *fieldName*, *instanceId*), which uses the matrix to return the index of the value in the domain for the given composite type, field name, and instance id. We then use this index to find the value in the Domain associated to the type of the field (getType($\breve{\theta}$, *fieldName*)). We define a function getValue($Type$, $index$), which returns the value from the Domain. These functions are defined to use look-up tables. The look-up tables are generated dynamically from the original finitization method of the target type.

3.1.2 Migration Procedure. To migrate a set of classes written for Korat to our format, we have to transform Java classes to our Types. For each Java class we have one Composite Type, and for each field that has a primitive type we have one Primitive Type. For simplicity of exposition, we use names of fields to name Primitive Types. In the next step, we use newly introduced Types to write finitization, i.e., specify the Domain for each Type.

In the final step of our migration, we convert field accesses (inside the `repOK` method) to use the matrix. First, we replace all uses of `null` for types in $\breve{\theta} \in \breve{\Theta}$ with `0`. Then, we rewrite all local variable declarations that use a type $\theta \in \Theta$ to use the type `int`. Next, we use the rewrite rules shown in Figure 6 to translate field accesses on objects to accesses to the matrix using the getValue and getIndex functions described earlier. Recall that `repOK` does not write to fields of the object. Therefore, our rules include only R, i.e., reads from fields. When applying these rewrite rules, the environment Γ contains the types of all expressions.

In the first rule (RDomainValue), accesses to variables that are now of type `int`, i.e., contain domain values, are left unchanged. Previously, they were a type $\theta \in \Theta$. If these variables originally referenced objects of a composite type, they will now contain the instanceId of those objects. The second rule (RImplFieldRef) rewrites implicit accesses to fields of the target type; the instanceId of type τ is always 1. The third rule (RFieldRef) rewrites field accesses to objects not of the target type; the rule must look up the instanceId of the object at runtime.

Since `repOK` can call other methods that operate on the object graph, we must also translate these methods to use the matrix. Let g be a function transitively reachable from `repOK`. We create a function g' from g that is initially identical in every way except the name. Next, we translate the return type and the types of arguments of g', replacing $\theta \in \Theta$ with `int`. Rewrite all field accesses in g' as it was done in `repOK`. Finally, change all references to g in `repOK` (and transitively reachable functions) to g'.

In our implementation, the getIndex function also tracks accesses to fields, which we need to implement the accessed-field optimized exploration (see Section 2.1). Each invocation of getIndex function records, into an internal list, the composite type, field name, and instance id that are given as the arguments.

3.2 Worklist Algorithm

Our worklist algorithm is the second necessary step to enable the bounded exhaustive testing on GPUs. The algorithm is used to overcome memory limitations of GPUs, because the existing GPUs either do not have virtual memory or the use of virtual memory is highly inefficient. This step is illustrated in Figure 10c (in the Appendix). Our algorithm automatically manages memory transfers between GPU and CPU without requiring any assistance by the users.

```
Require: ds data structure
Require: size desired data structures size
 1: function Cpu(ds, size)
 2:     overflow ← AllocateMainMemory(+∞)
 3:     in′ ← AllocateOnGpu(IN_SIZE)
 4:     out′ ← AllocateOnGpu(OUT_SIZE)
 5:     Add(ZeroCandidate(ds, size), in′)
 6:     while not IsEmpty(in′) do
 7:         numOfThreads ← Size(in′)
 8:         Gpu<<numOfThreads>>(in′, out′, ds, size)
 9:         outSize ← Size(out′)
10:         gpuToGpu ← Min(outSize, IN_SIZE)
11:         Move<<gpuToGpu>>(out′, in′, gpuToGpu)
12:         gpuToCpu ← outSize − gpuToGpu
13:         Move(out′, overflow, gpuToCpu)
14:         if gpuToGpu < IN_SIZE then
15:             missing ← IN_SIZE − gpuToGpu
16:             bufSize ← Size(overflow)
17:             cpuToGpu ← Min(missing, bufSize)
18:             Move(overflow, in′, cpuToGpu)
19:         end if
20:     end while
21: end function
```

Fig. 7. Algorithm for the process executed on CPU, which starts GPU kernels and manages memory usage

Figures 7 and 8 show the CPU and GPU parts of our worklist algorithm respectively. The CPU part is primarily concerned with launching the GPU kernel and managing the worklist so that large instances can be tested in the limited memory available on a GPU. The GPU part extracts work (i.e. candidate vectors) from the worklist, applies `repOK` to each candidate vector in parallel and generates additional candidate vectors based on the PKorat algorithm. The `while` loop from the PKorat algorithm on Line 4 (Figure 8) highlights one of the main challenges – it is an irregular loop with an unknown number of iterations so the number of candidate vectors it produces is not known in advance or even at runtime and consequently it is executed serially by each thread.

The input to the CPU algorithm (Figure 7) are the name of the data structure to generate and size of the desired instances. First, the algorithm allocates an "overflow" buffer in main memory to store candidate vectors that do not fit in the GPU memory (line 2). The size of this buffer is dynamic. Second, we allocate two worklists, `in'` and `out'` in GPU memory (lines 3 and 4). The `in'` worklist stores candidates to be processed in the next invocation of a GPU kernel. New candidate vectors are generated by the GPU kernel processing these input candidates and stored in the `out'` worklist. The size of these worklists is computed dynamically (at the startup) based on the amount of memory available on the GPU. Finally, we generate the initial candidate vector and store it in the input list (line 5) before entering the main loop of the algorithm.

The main loop of the algorithm iterates until the input worklist is empty. In each loop iteration, we launch a GPU kernel choosing the number of threads so that each thread will handle one candidate (line 7). The GPU kernel will generate the next set of candidates and store them into the output worklist (line 8); we wait for this invocation to finish. The maximum size of the input worklist (IN_SIZE) is chosen so that we will never run out of memory when storing the candidate vectors in the output worklist. Therefore, if the output worklist contains more candidates than IN_SIZE, we only move the first IN_SIZE candidates to the input worklist (line 11). Any excess

```
Require: in′ input worklist
Require: out′ output worklist
Require: ds data structure
Require: size desired data structures size
 1: function GPU(in′, out′, ds, size)
 2:     cv ← GET(threadId, in′)
 3:     isValid, accFields ← REPOK(ds, cv)
        ▷ Get next candidates with PKorat algorithm
 4:     while SIZE(accFields) > 0 ∧ PEEK(accFields) == 0 do
 5:         field ← POP(accFields)
            ▷ See [Siddiqui and Khurshid 2009] for NONISOMAX
 6:         for i ← 1, NONISOMAX(cv, accFields, field) do
 7:             cv[field] = i
 8:             atomic
 9:                 ADD(cv, out′)
10:             end atomic
11:         end for
12:         cv[field] = 0
13:     end while
14: end function
```

Fig. 8. Algorithm for threads that run on a GPU. Each thread evaluates one candidate vector, finds next set of candidates based on PKorat algorithm [Siddiqui and Khurshid 2009], and includes next set of candidates to the output worklist

candidates are moved from the GPU to the CPU overflow buffer (line 13). When the input worklist contains fewer than IN_SIZE items, we move items from the CPU overflow buffer back to the input worklist (lines 14–19).

Figure 8 shows the GPU kernel, which was invoked on line 8 from CPU. This code is executed in an single-program multiple data (SPMD) style, so all threads execute the same code in parallel. Each thread takes as input the two worklists (which are always in GPU memory), data structure to generate, and size of desired instances. Using CUDA-provided variables, each thread establishes its identity (`threadId`) and uses it to retrieve a candidate from the input list. Each thread therefore works on a different candidate in parallel.

Each thread now evaluates the predicate (i.e., `repOK`) on its candidate, which returns a boolean value to indicate if the candidate is valid, as well as the list of accessed fields (iff we used accessed-field optimized exploration). In case of accessed-field optimized exploration, we use PKorat algorithm [Siddiqui and Khurshid 2009] to find all candidate vectors, based on the accessed fields, which need to be explored. As a thread discovers new candidates, it adds them to the output worklist. Since many threads are executing in parallel, candidates are added atomically to the output worklist.

3.3 GPU-specific Optimizations

This section describes several steps to *optimize the exploration* by transforming the predicates and the exploration engine. We leverage the knowledge of predicates' structure to choose the appropriate data structures and memory structure to exploit the full potential of GPUs.

We will use our running example (`Binary Tree`) to illustrate the benefits of each optimization step as shown in Table 2; level 0 is the initial implementation enabled for GPUs (i.e., includes the encoding and the worklist algorithm), *Size* column shows the size of generated instances, *Time* shows the exploration time, and *Speedup* shows the speedup (over level 0) obtained with various optimizations; each optimization level includes a unique transformation (as described in the following sections) and includes all the optimizations from all previous optimization levels.

3.3.1 Removing Memory Allocations from the Exploration Engine (level 1). CUDA features like managed memory, C++11 compatibility, and support for dynamic memory allocation (`malloc`, `new`) [NVIDIA 2016] make it easy to take the existing C code and run it on the GPU with only minor modifications. Unfortunately, the resulting code can be very slow, often slower than single-threaded CPU. To obtain good performance, we removed function calls related to memory management from the exploration engine. These function calls included allocations of candidate vectors and domains. Rather than managing memory on the device, we preallocate the memory for each thread prior to invoking a kernel. This approach is feasible because the size of each candidate vector is constant, as well as the size of each type domain. **Running example:** Figure 11a (in the Appendix) illustrates this optimization step. This optimization step leads to 2.10× speedup (computed on average across all sizes) for our running example (Table 2).

Table 2. Test Generation Times and Speedups for Binary Tree with Different Optimization Levels

Level	Size	Time [s]	Speedup
0	12	16.65	1.00
0	13	65.82	1.00
0	14	256.03	1.00
1	12	7.75	2.14
1	13	30.75	2.14
1	14	122.29	2.09
2	12	0.55	30.27
2	13	2.34	28.12
2	14	10.32	24.80
3	12	0.27	61.66
3	13	1.10	59.83
3	14	4.53	56.51
4	12	0.23	72.39
4	13	0.92	71.54
4	14	3.73	68.64

3.3.2 Removing Memory Allocation from Predicates (level 2). Similar to the previous step, we transform predicates to remove functions that manage memory. These functions allocate space for data structures used to keep track of the visited nodes, various queues, etc. As in the previous step, the size of each of the data structures is bounded, so we preallocate memory for each thread prior to the kernel invocation. **Running example:** This transformation step is illustrated in Figure 11b (in the Appendix) and the optimization leads to 25.62× speedup for our running example (on average across all sizes in the Table 2).

3.3.3 Utilizing GPU Memory Structure for Fast Accesses (level 3). GPUs also support special memory structures such as constant memory that is designed to store read-only data shared by all the threads. It can be used to store coefficients for image filters, for example. Constant memory can only store up to 64KB of data. We use constant memory to store the values for type domains. Additionally, we transform code to use local memory for data structures in predicates, e.g., a set of visited nodes. **Running example:** These transformations are illustrated in Figure 11c (in the Appendix) and the optimized code is now 57.37× faster (on average across all sizes in the Table 2).

3.3.4 Using Well-known Operator Transformations for Fast Computation. Inspired by prior research findings, we use bitsets wherever applicable. GPUs do not support integer modulo operator and integer division in hardware. We therefore transformed expressions with these math operators to semantically equivalent expressions that are more efficient on GPUs (e.g., modulo operator was changed to several expressions that use + and - operators). Furthermore, if the size of generated instances permits, which we can find from the finitization prior to kernel invocation, our transformation uses bit sets and bit operations instead of int arrays for some data structures, e.g., set of visited nodes. **Running example:** Figure 11d (in the Appendix) illustrates this transformation step and Table 2 shows that some additional speedup can be obtained.

3.4 intKorat Variants

We have implemented intKorat for three different platforms: Java, C, and CUDA. For Java and C we implement only the proposed encoding, and for CUDA we implemented the encoding, the worklist algorithm, and the GPU-specific optimizations.

Korat^j and Naive^j. We implemented IntKorat in Java by closely following the existing implementation of Korat [Boyapati et al. 2002; Milicevic et al. 2007]. As we already described, unlike for Korat, users of Korat^j would specify properties directly on the candidate vectors rather than on object graphs. However, the users can still write the predicates for the original Korat and then use our automated migration procedure (Section 3.1) to obtain predicates that run on Korat^j. Korat^j was a natural transition step towards our tool for the C language. We also obtain an implementation, named Naive^j, which performs the naive exploration, by simply disabling the accessed-field optimization in Korat^j.

Korat^c and Naive^c. We implemented IntKorat in C by closely following Korat^j. Most of the challenges were related to mapping from Java constructs to C constructs. For example, we do not use any library in our implementation, but we implement everything that is necessary for predicates, e.g., a set that keeps visited nodes. Korat^c was a natural step towards our implementation for CUDA. We also obtained an implementation, named Naive^c, which performs the naive exploration, by simply disabling the accessed-field optimization in Korat^c.

Korat^g and Naive^g. We implemented IntKorat using CUDA 8. Our implementation includes the novel encoding presented in Section 3.1, the worklist algorithm presented in Section 3.2, as well as the GPU-specific optimizations described in Section 3.3.

4 EVALUATION

To evaluate the benefits of IntKorat (all three variants) we answer the following research questions:

- **RQ1:** What is the speedup obtained with IntKorat on GPUs when we explore all candidate vectors (i.e., Naive^g) over the original Korat?
- **RQ2:** What is the speedup obtained by test generation with IntKorat on GPUs when we use the accessed-field optimized exploration (i.e., Korat^g) over the original Korat?
- **RQ3:** Can Korat^g be easily run on different GPUs and how does the configuration of a GPU impact the benefits of Korat^g?
- **RQ4:** How does the encoding of predicates impact the test generation time (in various programming languages)?
- **RQ5:** How does the number of GPU threads impact the speedup of Korat^g?
- **RQ6:** What is the complexity of predicates for different encoding approaches and various programming languages?

We have run all the experiments on a machine with Intel Core i7-6700 3.40GHz CPU and 16 GB RAM. We used NVIDIA GTX 1080 GPU running at 1.73GHz with 8GB RAM. We used Java 1.8.0, GCC 4.8.4 (with -O3), and CUDA/NVCC 8.0. We used three other GPUs to answer RQ3; we describe these three additional GPUs in Section 4.4.

We briefly describe the data structures used in our experiments and then answer each research question in turn.

4.1 Data Structures

We evaluated various implementations of IntKorat using 7 data structures, which were commonly used in prior studies on test input generation [Boyapati et al. 2002; Kuraj et al. 2015; Sharma et al. 2011; Siddiqui and Khurshid 2009]. Table 3 (the first column) shows the list of the used data structures: bt - binary tree, bst - binary search tree, sll - singly linked list, ha - heap array, dll - doubly linked list, ds - disjoint set, and rbt - red black tree.

Table 3. Number of Total and Valid Candidate Vectors, Total Execution Time, and Speedup Over Naive^o when Generating Tests for Different Structures and Sizes using Naive Exploration

	Size	#Candidates		Naive^o	Naive^j		Naive^c		Naive^g	
		Total	Valid	Time [ms]	Time [ms]	Speedup	Time [ms]	Speedup	Time [ms]	Speedup
bt	3	16384	30	89	6	14.83	200	0.44	1464	0.06
bt	4	1953125	336	365	84	4.34	155	2.35	812	0.44
bt	5	362797056	5040	37671	12312	3.05	29412	1.28	1805	20.87
bt	6	96889010407	95040	10658168	4201802	2.53	8331927	1.27	140970	75.60
bst	3	442368	30	193	31	6.22	350	0.55	787	0.24
bst	4	500000000	336	52327	18743	2.79	39104	1.33	1852	28.25
bst	5	1133740800000	5040	126157143	48368003	2.60	93387280	1.35	750203	168.16
sll	5	279936	21576	135	15	9.00	150	0.90	818	0.16
sll	6	5764801	355081	575	130	4.42	313	1.83	805	0.71
sll	7	134217728	6805296	10824	2619	4.13	7199	1.50	954	11.34
ha	7	306110016	15221802	19963	2979	6.70	12750	1.56	923	21.62
ha	8	8100000000	317773557	493815	78293	6.30	314230	1.57	3701	133.42
ha	9	235794769100	7477128472	14725547	2310804	6.37	8995270	1.63	534347	27.55
dll	4	1953125	24	300	51	5.88	940	0.31	816	0.36
dll	5	362797056	120	27282	6418	4.25	15500	1.76	1674	16.29
dll	6	96889010407	720	7549669	1671807	4.51	4133601	1.82	118349	63.79
ds	3	314928	22866	153	21	7.28	170	0.90	810	0.18
ds	4	419430400	18288344	30270	7443	4.06	11164	2.71	1831	16.53
ds	5	1098632812500	32224753570	79923953	16452094	4.85	27898707	2.86	968504	82.52
rbt	2	78732	4	125	11	11.36	400	0.31	817	0.15
rbt	3	536870912	18	40593	10621	3.82	20938	1.93	1972	20.58
$\sum$	N/A	N/A	N/A	239729160	73144287	N/A	143199760	N/A	2534214	N/A

Source code distribution of Korat already includes all these data structures [Korat Home Page 2017]. However, we had to implement these data structures and all their predicates for Korat^j, Korat^c, and Korat^g. As discussed earlier, we can obtain the predicates for Korat^j via our automated migration procedure. (Semi-)automated procedure can also be developed to migrate from Korat^c to Korat^g, but we leave this for future work.

4.2 Naive Exploration on GPUs

RQ2: What is the speedup obtained with INTKORAT *on GPUs when we explore all candidate vectors (i.e., Naive^g) over the original Korat?*

We measured the time to naively explore *all candidate vectors* on GPUs (Naive^g), and we compared the results with the naive exploration using Naive^o (i.e., the original Korat implementation that does the naive exploration), Naive^j, and Naive^c. Specifically, in each run of our experiments, we set the bounds for one of the data structures and explore all candidate vectors to detect those that are valid. This was interesting, because the naive exploration is embarrassingly parallel, and we used it to discover necessary changes to our code to make it optimal for the execution on GPUs.

Table 3 shows, for each data structure and various sizes (the second column) the total number of candidate vectors (the third column) and the number of valid candidate vectors (the fourth column). Additionally, the Table 3 shows time (in milliseconds) to explore all candidate vectors for Naive^o, Naive^j, Naive^c, and Naive^g. Differences in the exploration time across several runs were negligible. We measured the execution time *from the start of each process until its completion*; this means that time for Naive^g includes, not only the time to execute GPU kernels but also time spent on CPU

computation and memory transfer. We also show speedup of our implementations over Naive^o (i.e., Naive^o/Naive^j, Naive^o/Naive^c, and Naive^o/Naive^g).

Our results show that, for *larger* sizes, Naive^g is always faster than Naive^j, Naive^j is always faster than Naive^c, and Naive^c is always faster than Naive^o. In other words, all our implementations outperform the naive exploration implemented in Java that evaluates predicates on in-memory object graphs. We can observe that Naive^g and Naive^c are the slowest for small data structures; this was not surprising due to the startup cost. However, Naive^c becomes more efficient than Naive^o, and Naive^g outperforms all other implementations for larger sizes (across all used data structures).

In sum, our Naive^g implementation (Section 3.4) speeds up naive test generation, on average, by 32.80× over Naive^o (20.80× over Naive^c and 8.92× over Naive^j).

4.3 Optimized Exploration on GPUs

RQ2: What is the speedup obtained by test generation with IntKorat *on GPUs when we use the accessed-field optimized exploration (i.e., Korat^g) over the original Korat?*

We measured the time to explore candidate vectors using the optimized algorithm with worklist (Section 3) on GPUs, and we compared the results with Korat^o, Korat^j, and Korat^c. Similar to the setup described in the previous section, we explore one data structure and one size at a time.

Table 4 shows, for various sizes of each data structure, the total number of predicates explored and the number of valid candidates. Note that sizes of target instances in Table 4 are substantially larger than target sizes in Table 3. Unlike for the naive exploration, there is no easy way to analytically compute the total and valid numbers of explored candidates, and we check the correctness of our exploration by comparing it to the original Korat; all implementations should give the same results.

Table 4 shows time (in milliseconds) to generate tests. Differences in the exploration time across several runs were negligible. As in the previous section, the first two columns show the name of a data structure and size, respectively. The times reported for Korat^o, Korat^j, Korat^c, and Korat^g are the total time for each process, as noted earlier. We also note speedups over Korat^o.

Our results show that Korat^g is more efficient than any other implementation for all data structures except sll and dll. The time to generate tests for sll and dll is negligible and the setup cost in these cases for Korat^g is higher than the generation time; this is not surprising. We included these results for completeness and comparison with the naive exploration.

Across all data structures Korat^g speeds up test generation, on average, 17.46× over Korat^o (10.47× over Korat^j and 22.35× over Korat^c). It is interesting to observe that the ratio of Naive^o/Naive^g is higher than Korat^o/Korat^g. This result was expected because Naive^g is embarrassingly parallel and code executed on GPU for Naive^g is simpler than code for Korat^g (i.e., accessed fields are not tracked for Naive^g, which results in much simpler code). At the same time, Korat^g is able to generate substantially larger instances than Naive^g.

4.4 Results on Various GPUs

RQ3: Can Korat^g be easily run on different GPUs and how does the configuration of a GPU impact the benefits of Korat^g ?

To demonstrate that our algorithm and implementation is not fine-tuned to a specific GPU, we evaluated Korat^g on total of four different GPUs. Table 5 shows the basic characteristics of these GPUs. For each GPU, we show the number of streaming multiprocessors (SM), the memory size, the raw memory bandwidth, and the CUDA capability (CC) which denotes the hardware GPU version. Each SM contains 2048 threads, so the number of hardware threads ranges from 26,624 (Quadro M4000) to 40,960 (GTX 1080).

We have used the same data structures and same sizes as in Section 4.3, i.e., we only evaluated optimized version of the exploration (i.e., Korat^g). We run Korat^g on all GPUs without any change.

Table 4. Number of Total and Valid Candidate Vectors, Total Execution Time, and Speedup Over Korat^o when Generating Tests for Different Structures and Sizes using Accessed-field Optimized Exploration

	Size	#Candidates		Korat^o	Korat^j		Korat^c		Korat^g	
		Total	Valid	Time [ms]	Time [ms]	Speedup	Time [ms]	Speedup	Time [ms]	Speedup
sll	32	1617	33	122	7	17.42	0	0.00	17	7.17
sll	63	6112	64	145	11	13.18	400	0.36	89	1.62
sll	126	24130	127	187	32	5.84	280	0.66	532	0.35
dll	16	453	1	117	5	23.40	0	0.00	7	16.71
dll	32	1677	1	129	8	16.12	100	1.29	25	5.16
dll	63	6234	1	153	21	7.28	140	1.09	133	1.15
bt	13	47795450	742900	25802	17955	1.43	31990	0.80	880	29.32
bt	14	186197720	2674440	100630	75461	1.33	135250	0.74	3613	27.85
bt	15	726257615	9694845	442634	314354	1.40	573598	0.77	15328	28.87
bst	9	20086300	4862	8581	5824	1.47	12985	0.66	487	17.62
bst	10	155455872	16796	71618	48763	1.46	110809	0.64	4062	17.63
bst	11	1206511438	58786	598225	413516	1.44	937606	0.63	35338	16.92
ha	9	51460480	10391382	6086	5396	1.12	11979	0.50	476	12.78
ha	10	583317405	111511015	72500	64327	1.12	150340	0.48	5348	13.55
ha	11	6913561920	1533143860	900796	810760	1.11	1956450	0.46	67831	13.28
ds	5	413855	41546	462	140	3.30	188	2.45	9	51.33
ds	6	33436639	2967087	14451	8171	1.76	18428	0.78	448	32.25
ds	7	3819937166	304142120	1726556	1100161	1.56	2673944	0.64	56107	30.77
rbt	9	1510006	122	1817	1037	1.75	2820	0.64	96	18.92
rbt	10	7530712	260	6180	5471	1.12	11501	0.53	482	12.82
rbt	11	39089158	586	30740	28686	1.07	65243	0.47	2878	10.68
$\sum$	N/A	N/A	N/A	4007931	2900106	N/A	6694051	N/A	194186	N/A

Table 5. Basic Characteristics of Various GPUs Used in Our Experiments. All GPUs Support 2048 Threads/SM. Legend: CC=Compute Capability, B/W=Bandwidth in Gigabyte/s

GPU	#SM	RAM (B/W)	CC	Introduced
GeForce GTX 1080	20	8GB (305)	6.1	2016
Tesla K80	13	12GB (240)	3.7	2014
Quadro M4000	13	8GB (192)	5.3	2015
Tesla K40c	15	12GB (288)	3.5	2013

Table 6 shows absolute time (Korat^g) and speedups (Korat^o/Korat^g) for each GPU (two columns per GPU). Absolute time and speedups for GTX 1080 are replicated from the Korat^g column in Table 4 to simplify the comparison.

Our results show that Korat^g is efficient regardless of GPU characteristics. Additionally, we can order GPUs based on the average speedup (from higher to lower): GeForce GTX 1080, Tesla K80, Tesla K40c, and Quadro M4000. This ordering corresponds to ordering these GPUs by their memory bandwidth, except that the K80 outperforms the K40 for a large number of structures despite having 20% less memory bandwidth. The K80 has 2× the number of physical registers as the K40 which reduces register spills and consequently local memory traffic in complex `repOK` implementations

Table 6. Korat^g Speedup Over Korat^o on Different GPUs

Structure	Size	GeForce GTX 1080		Tesla K80		Quadro M4000		Tesla K40c	
		Time [ms]	Speedup	Time [ms]	Speedup	Time [ms]	Speedup	Time [ms]	Speedup
sll	32	17	7.17	25	4.88	33	3.69	19	6.42
sll	63	89	1.62	117	1.23	181	0.80	93	1.55
sll	126	532	0.35	595	0.31	1235	0.15	596	0.31
dll	16	7	16.71	10	11.70	14	8.35	11	10.63
dll	32	25	5.16	31	4.16	54	2.38	35	3.68
dll	63	133	1.15	144	1.06	275	0.55	167	0.91
bt	13	880	29.32	1503	17.16	2233	11.55	1958	13.17
bt	14	3613	27.85	6204	16.22	9464	10.63	8223	12.23
bt	15	15328	28.87	26172	16.91	41107	10.76	34429	12.85
bst	9	487	17.62	648	13.24	736	11.65	696	12.32
bst	10	4062	17.63	5409	13.24	6211	11.53	5875	12.19
bst	11	35338	16.92	46510	12.86	52954	11.29	50698	11.79
ha	9	476	12.78	616	9.87	843	7.21	767	7.93
ha	10	5348	13.55	7158	10.12	9951	7.28	8943	8.10
ha	11	67831	13.28	90686	9.93	124036	7.26	113047	7.96
ds	5	9	51.33	15	30.80	20	23.10	18	25.66
ds	6	448	32.25	793	18.22	1156	12.50	934	15.47
ds	7	56107	30.77	99019	17.43	148076	11.65	118479	14.57
rbt	9	96	18.92	145	12.53	150	12.11	141	12.88
rbt	10	482	12.82	752	8.21	770	8.02	722	8.55
rbt	11	2878	10.68	4351	7.06	4355	7.05	4090	7.51

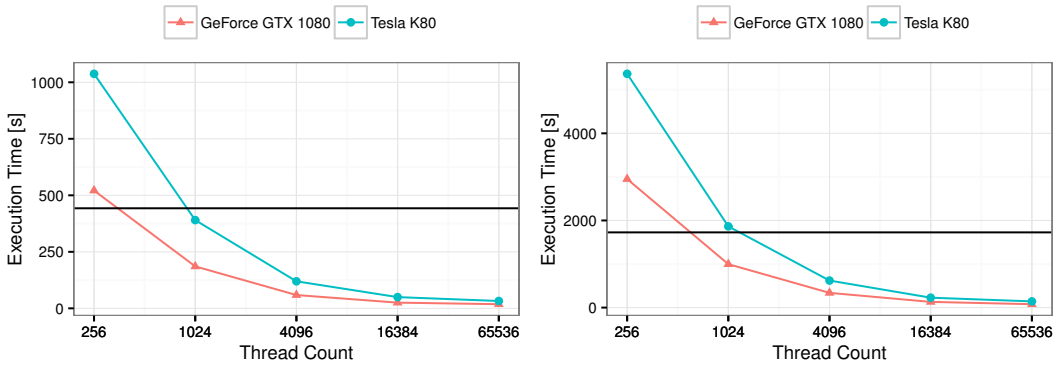
to balance out the lower memory bandwidth; recall that we use registers and local memory for several data structures inside predicates.

4.5 Impact of Encoding on Test Generation

RQ4: How does the encoding of predicates impact the test generation time (in various programming languages)?

It is interesting to discuss the differences in the results for sequential implementations: Korat^o, Korat^j, and Korat^c. We will once again consider the results in Tables 3 and 4.

Naive^j was clearly the fastest approach among the three sequential approaches for naive exploration. Also, Naive^c was faster than Naive^o for larger sizes across all structures. Based on our results for the naive exploration, we can conclude that writing predicates that directly use candidate vectors provides substantial benefits: Naive^j showed 5.68× speedup over Naive^o, and Naive^c showed 1.43× speedup over Naive^o. We suspect that Naive^j is faster than Naive^c due to the Just in Time Compilation (JIT) available in Java [Savija T. V. 2011, Understanding Just-in-Time Compilation and Optimization]. We tried to disable JIT compiler (`-Xint`) and the results showed that Naive^c outperforms Naive^j. We also tried to use profile-guided optimization with GCC (the first compiler run was with `-march=native -fprofile-generate` and the subsequent compiler run was with `-march=native -fprofile-use`). However, we observed insignificant differences



(a) Binary tree (bt) (b) Disjoint set (ds)

Fig. 9. Total test generation time for bt and ds with Korat^g using different number of threads. The horizontal (black) line shows test generation time with Korat^o

in performance when using profile-guided optimization. Although further investigation can be performed to understand the differences between Naive^j and Naive^c, this is outside of the scope of this paper.

Regarding the accessed-field optimized exploration, our results show (Table 4) that there is no clear winner among Korat^o, Korat^j, and Korat^c. This happens because Korat^j and Korat^c have complex predicates that behave in data-dependent fashion making it hard for compilers to statically optimize or parallelize code. The presence of irregular data-dependent `while` loops makes even runtime parallelization hard. Still, we find that either Korat^j or Korat^c is faster than Korat^o in most cases. In the future, it will be interesting to explore implementations on systems that support parallelization of such irregular algorithms [Kulkarni et al. 2007].

4.6 GPU Thread Usage

RQ5: How does the number of GPU threads impact the speedup of Korat^g?

Figure 9 shows the execution time for Korat^g for bt and ds as the number of threads used in the kernel invocation is varied. Other data structures follow a similar pattern. The effect of decreasing the number of threads is that GPU memory requirement for the worklists is reduced which is useful for GPUs with limited memory. However, using fewer threads can also underutilize the GPU. We note that on the two fastest used GPUs, reducing the number of threads to below what can be concurrently executed (40,960 for the GeForce GTX 1080 and 30,720 for the Tesla K80) has a negative impact on the performance.

4.7 Complexity of Predicates

RQ6: What is the complexity of predicates for different encoding approaches and various programming languages?

We measured the complexity of predicates by collecting the number of lines of code (LOC) for `repOK`, as well as McCabe Code Complexity metric (MCCC) [Ammann and Offutt 2008]. We used SourceMeter (https://www.sourcemeter.com) to collect both LOC and MCCC. We measured LOC and MCCC not only for `repOK`, but (transitively) included all functions that are related to the predicates. Table 7 shows our results. We can observe that `repOK` in Korat^o is usually shorter than `repOK` in Korat^j, which is in turn shorter than code in Korat^c. (SourceMeter was not able to collect metrics for CUDA files, but we manually confirmed that predicates for Korat^g have more LOC than any other implementation.)

Table 7. Lines of Code (LOC) and McCabe Code Complexity (MCCC) Metrics of repOK Methods for the Original Korat and intKorat Implementations

	Korat^o		Korat^j		Korat^c	
Structure	LOC	MCCC	LOC	MCCC	LOC	MCCC
sll	26	9	22	3	27	4
dll	29	10	39	8	47	8
bt	23	7	49	8	59	8
bst	52	24	84	17	107	18
ha	20	10	33	9	35	9
ds	35	17	70	17	75	17
rbt	90	33	204	46	229	47

We can observe one case (sll) when code in Korat^o has more lines than Korat^j; this happened because the original Korat developers split `repOK` into several methods (which was probably needed only during their experiments). Unlike LOC, it is interesting to observe that MCCC shows that Korat^o can be more complex than Korat^c in several cases. This happens because in our implementation of Korat^c we avoid recursive function calls to enable easier migration to Korat^g, as stack sizes available on GPUs are limited [NVIDIA 2016].

Our current evaluation of the complexity is based on quantitative data. Based on our experience, (re-)writing predicates to use candidate vectors is slightly more challenging than writing the original predicates; our automated translation procedure (Section 3.1) makes the migration trivial.

5 DISCUSSION

Correctness. We checked the correctness of Naive^j, Naive^c, and Naive^g by comparing the numbers of total and valid candidates with those obtained with Naive^o. Similarly, we compared numbers for Korat^j, Korat^c, and Korat^g with Korat^o. Recall that Korat^o implementation is already publicly available [Korat Home Page 2017], and obtaining Naive^o from Korat^o is trivial. Additionally, for the naive exploration, the number of total and valid candidates can be computed with a simple formula, which we also used to confirm our results.

Memory limit. Our current algorithm (Section 3.2) assumes that enough main memory is available. However, due to high memory usage, we could observe thrashing [Denning 1968]. We could extend our algorithm to store candidate vectors on disk as necessary while limiting the number of candidates in the main memory (similar to how we balance the GPU memory and the main memory usage). In our evaluation there was no need for this extension.

We note that newly available virtual memory mechanisms on NVIDIA Pascal-based GPUs could be used to simplify our algorithm by eliminating the need to handle worklist overflow. These mechanisms allow us to allocate memory beyond GPU physical memory limits. The GPU memory then functions essentially as a cache, where pages are brought in only when referenced. In our experiments with this technique on the GTX 1080, we observed 2× slowdown compared to our technique for worklist overflow management for BT (size 14). Using the NVIDIA profiler, we could attribute all of the slowdown to these VM mechanisms. Since worklist read/write patterns are simple and linear, implementations of GPU virtual memory must be significantly improved before they can outperform our manual overflow management.

Test generation on both CPU and GPU. Prior work on PKorat has explored the use of (a large number of) general purpose CPUs to speed up testing by distributing test generation. On the other hand, we parallelize test generation by utilizing GPUs. In the future, we plan to combine these two approaches. There are several directions that we plan to explore. First, we can generate tests both on CPU and GPU in parallel on a single machine. Second, we can utilize multiple CPUs and GPUs on a cluster of machines.

Lock-free worklist. GPUs do not support atomic sections and spinlocks, therefore we implement a lock-free worklist using atomic primitives. When executed by a thread, the pop function (Figure 8) of the worklist atomically increments a pointer into the worklist using `atomicInc` CUDA primitive which returns the previous value of the pointer. Thus, each thread receives a unique candidate from the worklist which is copied by the pop function into the local array provided by the thread. Similar, when a thread executes push, it provides a candidate vector stored in its local array, which is copied to a unique position in the worklist by the push function.

6 LIMITATIONS

Our results may not generalize to other data structures. To mitigate this threat, we used the common data structures used in prior work on test case generation [Boyapati et al. 2002; Gligoric et al. 2010; Kuraj et al. 2015; Siddiqui and Khurshid 2009]. We observed that Korat^g consistently provides speedup across all data structures, as long as the original test generation is costly. This result holds for both naive and accessed-field optimized exploration.

The results may differ for other types of CPUs and GPUs, as well as different versions of CUDA. To mitigate this threat, we ran our experiments on four different GPUs available in our lab and reported speedups. Based on the results, Korat^g consistently showed improvement over Korat^o.

NVIDIA GPUs can also be programmed using OpenCL [Khronos Group 2017] which is also supported by other GPUs such as those from AMD and ARM. We use CUDA because NVIDIA's toolchain only supports older versions of OpenCL and many tools such as profilers are not supported. In addition, our current CUDA implementation uses C++ heavily whilst the NVIDIA OpenCL implementation only supports C99 making it tedious to port our code. OpenCL version 2.2, released in May 2017, now mandates C++ support though it is still to gain widespread hardware support. Although a study from 2011 has shown no substantial differences in performance between CUDA and OpenCL [Fang et al. 2011], we note that both CUDA and OpenCL have evolved since then.

Our implementations of Korat^j, Korat^c, and Korat^g, as well as predicates of all data structures may contain bugs. We have already discussed a way that we used to check correctness of various implementations (Section 5). Additionally, we wrote unit tests, reviewed code and scripts, and reused existing scripting infrastructure.

We have compared Korat^g only to sequential implementations. To the best of our knowledge, Korat^g is the first bounded-exhaustive testing technique that runs on GPUs. Prior work on parallel testing either focuses on speeding up test generation and execution together [Misailovic et al. 2007] or requires specialized hardware [Misailovic et al. 2007; Siddiqui and Khurshid 2009]; we also know that PKorat led to 7.05× speedup. Unlike prior work, we enabled parallel test generation on commodity hardware.

7 RELATED WORK

There has been a lot of work on test input generation and algorithm optimizations using GPUs. We first present several approaches for test input generation. We then discuss various graph algorithms implemented on GPUs. Finally, we present prior work on constraint solving on GPUs and testing GPU kernels.

Bounded exhaustive testing. We have already discussed the original Korat and PKorat algorithms in detail throughout the paper [Boyapati et al. 2002]. Alloy (Analyzer) explores models up to a given small bound [Jackson 2006]. Others have studied the use of model checking tools to exhaustively generate data structures [Visser et al. 2006]. TestEra is Korat's predecessor where the users write predicates in a declarative language [Marinov and Khurshid 2001]. UDITA introduced a novel language that allows specification of properties by mixing predicates written in imperative and declarative style [Gligoric et al. 2010]. UDITA optimizes (sequential) test input generation with a delayed execution. Similar approach was followed by Rosner et al. [Rosner et al. 2014]. SciFe uses an expressive algebra of enumerators to make generation compositional [Kuraj et al. 2015]. All these approaches run on CPUs, and our work focuses on parallel execution of Korat algorithm on GPUs. In the future, we plan to explore optimizing other algorithms by running them on GPUs.

Parallel Korat was developed to optimize test generation and execution (together) on map-reduce model [Misailovic et al. 2007]. Initially, the space of candidates vectors is partitioned on a single machine and each partition is given to a different worker for further exploration and execution. Parallel Korat was evaluated on a Google cluster and substantial speedup was reported. Unlike parallel Korat, intKorat optimizes the enumeration of candidate vectors (which is equivalent to the first/sequential phase of Parallel Korat) rather than test execution. Recent work introduced *memoization* of some key pruning steps of the Korat search to optimize Korat's *re-execution* [Dini 2016; Dini et al. 2017]. Memoization is complementary to our use of GPUs and the two approaches can be applied in tandem.

Other test generation strategies. Many other (non-bounded exhaustive) test generation techniques have been proposed. Randoop is a feedback-directed test input generation [Pacheco et al. 2007; Randoop Home Page 2017]. Randoop builds a sequence of method calls by randomly choosing a method to invoke and the arguments for the invocation, and refines its generation process based on the execution of each sequence. Search-based test generation techniques use genetic algorithms to generate tests [Fraser and Arcuri 2013; McMinn 2011; Tonella 2004]. Most of these techniques generate the initial population of tests randomly and then improve test suite by evolving the initial population; search is guided by a fitness function, e.g., maximize code coverage. intKorat is a novel technique for bounded exhaustive testing. Parallelizing other test generation techniques is an interesting future direction.

Graph algorithms. Worklists and irregular loops feature in many implementations of graph algorithms on GPUs including some that implement compiler analyses [Mendez-Lojo et al. 2012; Prabhu et al. 2011]. Unlike PKorat, however, these irregular loops have a fixed iteration count which enables simpler inspector–executor approaches [Merrill et al. 2015] that can be automated using compilers [Pai and Pingali 2016]. Our implementation of intKorat uses CPU memory to work around limited GPU memory and consequently uses bulk-synchronous execution. A future implementation could use the Groute framework [Ben-Nun et al. 2017] to distribute execution of the algorithm asynchronously over multiple GPUs.

Constraint solving on GPUs. Researchers have recently proposed several techniques for (SAT) constraint solving on GPUs [Campeotto et al. 2014; Dal Palù et al. 2015]. Although Korat can be seen as a constraint solver, constraints are specified in an imperative language. Additionally, our work targets test input generation rather than constraint solving in general.

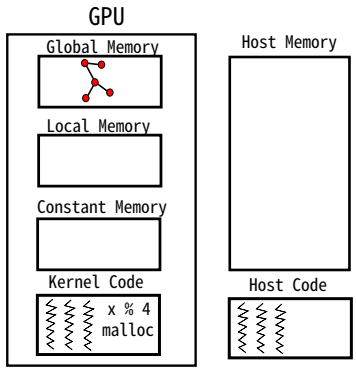
Analyzing, testing, and verifying GPU kernels. There has been a lot of recent work on analyzing, testing, and verifying GPU kernels [Betts et al. 2015; Boyer et al. 2008; Leung et al. 2012; Li and Gopalakrishnan 2010; Zheng et al. 2011]. Our work is complementary to these techniques as we use GPUs to speed up testing process.

8 CONCLUSIONS

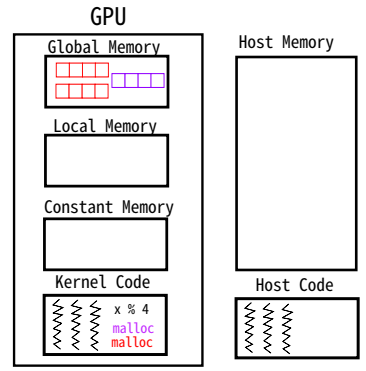
Testing is the most common approach in industry to check correctness of software. Automated test generation techniques help developers to quickly obtain high-quality test suites. Bounded exhaustive testing techniques generate all test cases up to the given bound that satisfy the given predicate. We presented the first bounded exhaustive technique, dubbed intKorat, which is suitable for run on GPUs. Unlike the existing techniques, which require predicates to be evaluated on in-memory object graphs, our technique requires predicates that operate on integer arrays that encode the object graphs. This encoding can speed up processing even if run sequentially. However, the full power of the encoding can be obtained if test generation is run on GPUs. Our results show that intKorat speeds up test generation, on average, by 17.46×.

A APPENDIX

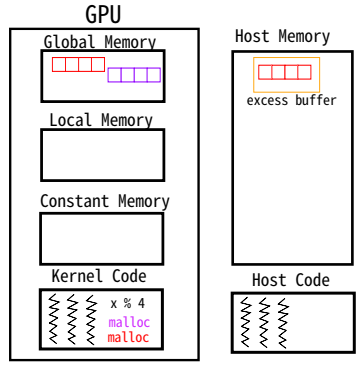
Figure 10 illustrates the way intKorat enables the test generation on GPUs. Figure 11 illustrates GPU-specific optimizations implemented in intKorat for CUDA (i.e., Korat^g).



(a) In the original Korat all objects are in the global memory and directly operated upon

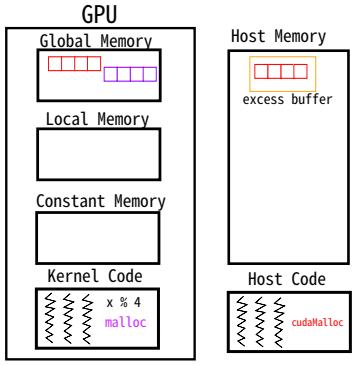


(b) In intKorat objects are encoded as integer arrays and then operated upon (Section 3.1)

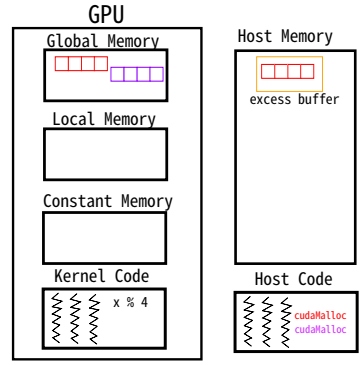


(c) In intKorat the excess buffer in host memory is used to save extra candidate vectors (Section 3.2)

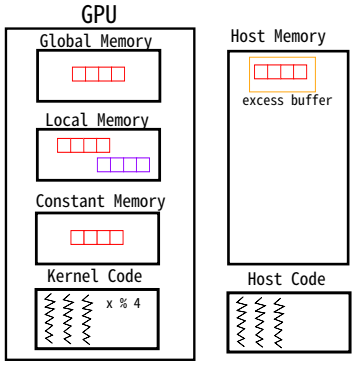
Fig. 10. Enabling exploration on GPUs



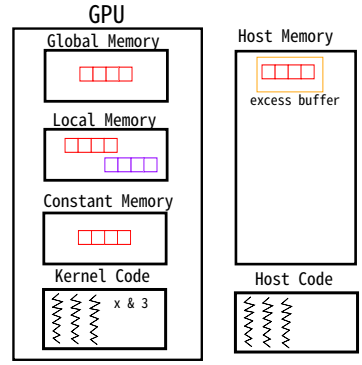
(a) Memory allocations are removed from the exploration engine



(b) Memory allocations are removed from the predicates



(c) Local and constant memory are used for fast accesses (to type domains)



(d) Bitwise operations and bit sets are used for efficient computation

Fig. 11. GPU-specific optimizations and their effect on memory structure (Section 3.3)

ACKNOWLEDGMENTS

We thank the anonymous reviewers for comments that improved this paper. Additionally, we thank Nima Dini, Sasa Misailovic, and Keshav Pingali for their feedback on this work. This research was partially supported by the US National Science Foundation under Grants Nos. CCF-1319688, CCF-1566363, CCF-1652517, and CNS-1239498.

REFERENCES

Paul Ammann and Jeff Offutt. 2008. *Introduction to Software Testing*. Cambridge University Press.

Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations. In *Symposium on Principles and Practice of Parallel Programming*. 235–248.

Antonia Bertolino. 2007. Software Testing Research: Achievements, Challenges, Dreams. In *Future of Software Engineering*. 85–103.

Adam Betts, Nathan Chong, Alastair F. Donaldson, Jeroen Ketema, Shaz Qadeer, Paul Thomson, and John Wickerson. 2015. The Design and Implementation of a Verification Technique for GPU Kernels. *ACM Trans. Program. Lang. Syst.* 37, 3 (2015), 10:1–10:49.

Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: Automated testing based on Java predicates. In *International Symposium on Software Testing and Analysis*. 123–133.

Michael Boyer, Kevin Skadron, and Westley Weimer. 2008. Automated dynamic analysis of CUDA programs. In *Workshop on Software Tools for MultiCore Systems*.

Federico Campeotto, Alessandro Palù, Agostino Dovier, Ferdinando Fioretto, and Enrico Pontelli. 2014. Exploring the Use of GPUs in Constraint Solving. In *International Symposium on Practical Aspects of Declarative Languages*. 152–167.

Alessandro Dal Palù, Agostino Dovier, Andrea Formisano, and Enrico Pontelli. 2015. CUD@SAT: SAT solving on GPUs. *Journal of Experimental & Theoretical Artificial Intelligence* 27, 3 (2015), 293–316.

Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. 2007. Automated Testing of Refactoring Engines. In *International Symposium on Foundations of Software Engineering*. 185–194.

Peter J Denning. 1968. Thrashing: Its causes and prevention. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*. ACM, 915–922.

Nima Dini. 2016. *MKorat: A Novel Approach for Memoizing the Korat Search and Some Potential Applications*. Master's thesis. University of Texas at Austin.

Nima Dini, Cagdas Yelen, and Sarfraz Khurshid. 2017. Optimizing Parallel Korat Using Invalid Ranges. In *International SPIN Symposium on Model Checking of Software*. 182–191.

A. G. Duncan and J. S. Hutchison. 1981. Using Attributed Grammars to Test Designs and Implementations. In *International Conference on Software Engineering*. 170–178.

Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. 2011. A Comprehensive Performance Comparison of CUDA and OpenCL. In *International Conference on Parallel Processing*. 216–225.

Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *Transactions on Software Engineering* 39, 2 (2013), 276–291.

Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. 2010. Test generation through programming in UDITA. In *International Conference on Software Engineering*. 225–234.

Daniel Jackson. 2006. *Software Abstractions: Logic, language, and analysis*. MIT Press.

Khronos Group. 2017. OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems. (2017). https://www.khronos.org/opencl.

Korat Home Page 2017. (2017). http://korat.sourceforge.net/index.html.

Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L Paul Chew. 2007. Optimistic parallelism requires abstractions. *ACM SIGPLAN Notices* 42, 6 (2007), 211–222.

Ivan Kuraj, Viktor Kuncak, and Daniel Jackson. 2015. Programming with enumerable sets of structures. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 37–56.

Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. 2010. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. In *International Symposium on Computer Architecture*. 451–460.

Alan Leung, Manish Gupta, Yuvraj Agarwal, Rajesh Gupta, Ranjit Jhala, and Sorin Lerner. 2012. Verifying GPU kernels by test amplification. *ACM SIGPLAN Notices* 47, 6 (2012), 383–394.

Guodong Li and Ganesh Gopalakrishnan. 2010. Scalable SMT-based verification of GPU kernel functions. In *International Symposium on Foundations of Software Engineering*. 187–196.

Barbara Liskov and John Guttag. 2000. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design* (1st ed.). Addison-Wesley Longman Publishing Co., Inc.

Darko Marinov and Sarfraz Khurshid. 2001. TestEra: A Novel Framework for Automated Testing of Java Programs. In *Automated Software Engineering*. 22–31.

Phil McMinn. 2011. Search-Based Software Testing: Past, Present and Future. In *International Conference on Software Testing, Verification and Validation Workshops*. 153–163.

Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. 2012. A GPU Implementation of Inclusion-based Points-to Analysis. In *Symposium on Principles and Practice of Parallel Programming*. 107–116.

Duane Merrill, Michael Garland, and Andrew Grimshaw. 2015. High-Performance and Scalable GPU Graph Traversal. *ACM Trans. Parallel Comput.* 1, 2 (2015), 30.

Aleksandar Milicevic, Sasa Misailovic, Darko Marinov, and Sarfraz Khurshid. 2007. Korat: A Tool for Generating Structurally Complex Test Inputs. In *International Conference on Software Engineering, Demo*. 771–774.

Sasa Misailovic, Aleksandar Milicevic, Nemanja Petrovic, Sarfraz Khurshid, and Darko Marinov. 2007. Parallel Test Generation and Execution with Korat. In *International Symposium on Foundations of Software Engineering*. 135–144.

Razieh Nokhbeh Zaeem and Sarfraz Khurshid. 2012. Test input generation using dynamic programming. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 34.

NVIDIA. September 2016. *CUDA C Programming Guide v8.0*. NVIDIA.

Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *International Conference on Software Engineering*. 75–84.

Sreepathi Pai and Keshav Pingali. 2016. A Compiler for Throughput Optimization of Graph Algorithms on GPUs. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 1–19.

Pablo Ponzio, Nazareno Aguirre, Marcelo F. Frias, and Willem Visser. 2016. Field-exhaustive Testing. In *International Symposium on Foundations of Software Engineering*. 908–919.

Tarun Prabhu, Shreyas Ramalingam, Matthew Might, and Mary Hall. 2011. EigenCFA: Accelerating Flow Analysis with GPUs. In *Symposium on Principles of Programming Languages*. 511–522.

Randoop Home Page 2017. (2017). https://github.com/randoop/randoop.

Nicolás Rosner, Valeria Bengolea, Pablo Ponzio, Shadi Abdul Khalek, Nazareno Aguirre, Marcelo F. Frias, and Sarfraz Khurshid. 2014. Bounded Exhaustive Test Input Generation from Hybrid Invariants. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 655–674.

Savija T. V. 2011. *Oracle JRockit Introduction, Release R28*. http://docs.oracle.com/cd/E15289_01/doc.40/e15058/underst_jit.htm.

Rohan Sharma, Milos Gligoric, Andrea Arcuri, Gordon Fraser, and Darko Marinov. 2011. Testing container classes: Random or Systematic?. In *Fundamental Approaches to Software Engineering*. 262–277.

Junaid Haroon Siddiqui and Sarfraz Khurshid. 2009. PKorat: Parallel Generation of Structurally Complex Test Inputs. In *International Conference on Software Testing, Verification, and Validation*. 250–259.

Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. November 2016. The Top500 List Statistics – Accelerator/Co-processor Share. (November 2016). https://www.top500.org/statistics/list/.

Kevin Sullivan, Jinlin Yang, David Coppit, Sarfraz Khurshid, and Daniel Jackson. 2004. Software Assurance by Bounded Exhaustive Testing. In *International Symposium on Software Testing and Analysis*. 133–142.

Kuo-Chung Tai and Yu Lei. 2002. A test generation strategy for pairwise testing. *Transactions on Software Engineering* 28, 1 (2002), 109–111.

Paolo Tonella. 2004. Evolutionary Testing of Classes. In *International Symposium on Software Testing and Analysis*. 119–128.

Willem Visser, Corina S. Păsăreanu, and Radek Pelánek. 2006. Test Input Generation for Java Containers Using State Matching. In *International Symposium on Software Testing and Analysis*. 37–48.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Conference on Programming Language Design and Implementation*. 283–294.

Mai Zheng, Vignesh T Ravi, Feng Qin, and Gagan Agrawal. 2011. GRace: A low-overhead mechanism for detecting data races in GPU programs. In *ACM SIGPLAN Notices*, Vol. 46. 135–146.