Design, Implementation, and Application of GPU-Based Java Bytecode Interpreters

AHMET CELIK, The University of Texas at Austin, USA
PENGYU NIE, The University of Texas at Austin, USA
CHRISTOPHER J. ROSSBACH, The University of Texas at Austin and VMware Research, USA
MILOS GLIGORIC, The University of Texas at Austin, USA

We present the design and implementation of GVM, the first system for executing Java bytecode entirely on GPUs. GVM is ideal for applications that execute a large number of short-living tasks, which share a significant fraction of their codebase and have similar execution time. GVM uses novel algorithms, scheduling, and data layout techniques to adapt to the massively parallel programming and execution model of GPUs.

We apply GVM to generate and execute tests for Java projects. First, we implement a sequence-based test generation on top of GVM and design novel algorithms to avoid redundant test sequences. Second, we use GVM to execute randomly generated test cases. We evaluate GVM by comparing it with two existing Java bytecode interpreters (Oracle JVM and Java Pathfinder), as well as with the Oracle JVM with just-in-time (JIT) compiler, which has been engineered and optimized for over twenty years. Our evaluation shows that sequence-based test generation on GVM outperforms both Java Pathfinder and Oracle JVM interpreter. Additionally, our results show that GVM performs as well as running our parallel sequence-based test generation algorithm using JVM with JIT with many CPU threads. Furthermore, our evaluation on several classes from open-source projects shows that executing randomly generated tests on GVM outperforms sequential execution on JVM interpreter and JVM with JIT.

CCS Concepts: • **Software and its engineering** → *Object oriented languages*; *Interpreters*; *Runtime environments*; *Software testing and debugging*.

Additional Key Words and Phrases: Java bytecode interpreter, Graphics Processing Unit, Sequence-based test generation, Shape matching, Complete matching

ACM Reference Format:
Ahmet Celik, Pengyu Nie, Christopher J. Rossbach, and Milos Gligoric. 2019. Design, Implementation, and Application of GPU-Based Java Bytecode Interpreters. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 177 (October 2019), 28 pages. https://doi.org/10.1145/3360603

1 INTRODUCTION

Graphics Processing Units (GPUs), are widely available nowadays on commodity hardware [Amazon 2018; Azure 2018; Google 2018]. Despite widespread availability and massive compute density, using GPUs for general purpose workloads remains challenging and is usually done by a few skilled programmers. Migrating an application to GPUs requires substantial change to the design of the application and fine-tuning to extract good performance. Code for GPUs is usually written

Authors' addresses: Ahmet Celik, The University of Texas at Austin, Austin, TX, 78712, USA, ahmetcelik@utexas.edu; Pengyu Nie, The University of Texas at Austin, Austin, TX, 78712, USA, pynie@utexas.edu; Christopher J. Rossbach, The University of Texas at Austin and VMware Research, Austin, TX, 78712, USA, rossbach@cs.utexas.edu; Milos Gligoric, The University of Texas at Austin, Austin, TX, 78712, USA, gligoric@utexas.edu.

2475-1421/2019/10-ART177
https://doi.org/10.1145/3360603

in low-level native languages (CUDA [NVIDIA 2019] or OpenCL [Khronos 2019]). Thus existing applications, particularly those written in a higher-level language such as Java, cannot benefit from GPUs out-of-the-box.

Several research [Catanzaro et al. 2010; Hayashi et al. 2013; Klöckner et al. 2012; Palkar et al. 2018; Prasad et al. 2011; Rossbach et al. 2013] and industrial projects [Gregory and Miller 2012; Oracle 2019d] have tried to *extend* high-level languages with support for GPUs. The goal of these projects was to speed up *parts of applications* that can benefit from data parallelism. Applications that manipulate streams or use explicit loops can benefit from parallel processing by moving computation to GPUs. These projects follow a two step workflow. First, a developer would manually modify code to expose fragments that are a good fit for a GPU by adding annotations [GPU 2019; Rossbach et al. 2013], extending specific classes [Pratt-Szeliga et al. 2012; Zaremba et al. 2012], or coding with GPU-friendly parallel patterns [Brown et al. 2011; Rossbach et al. 2013]. Second, those code fragments would be translated to CUDA or OpenCL either at *compile time* or at *runtime*.

Unfortunately, no prior work supports parallel execution of a large number of *independent Java processes* on GPUs; we assume that each process executes an independent task and uses various language features (e.g., object allocation, dynamic dispatch), native methods, and the Java Class Library (JCL). A system that enables this could accelerate various applications, including sequence-based test generation [Visser et al. 2006], execution of randomly generated tests [Pacheco et al. 2007], testing software product lines [Kim et al. 2013], symbolic execution [King 1976], software model checking [Godefroid 1997; Visser et al. 2003], etc. In other words, it would be an environment appropriate for Java processes that share code and execute similar sequences of instructions.

We present a design and implementation of GVM, the first system for running lightweight Java bytecode interpreters entirely on GPUs. Each Java bytecode interpreter, dubbed *TinyBee*, is executed by a single GPU thread. All TinyBees in the system share code (i.e., classfiles), but each TinyBee has its own heap, static area, operand stack, frame stack, and program counter. TinyBees support many features, including class loading, exception handling, dynamic dispatch, etc. Thus, TinyBees can execute a variety of applications, including those that manipulate objects on the heap or use the JCL. Moreover, TinyBees can handle native methods via an interface similar to JNI [Oracle 2019c] and MJI [Visser et al. 2003]. A significant challenge for running independent bytecode interpreters on a GPU is extracting good performance from the GPU's underlying SIMD execution model, which is designed for large numbers of threads that mostly follow the same control flow path. GVM targets large scale testing workloads that naturally have substantial common control flow during execution and execute in similar amount of time.

We applied GVM to generate and execute tests for Java projects. First, on top of GVM, we implemented the first parallel *systematic sequence-based test generation* technique. In a sequence-based test generation technique each generated test is a sequence of method calls in the system under test [Visser et al. 2006]. In our implementation, each TinyBee executes one unique sequence of method calls. We designed and developed two algorithms that explore different numbers of test sequences and use two approaches to avoid redundant method sequences. Our algorithms explore the same method sequences as existing sequential algorithms that run on a CPU, thus providing the same guarantees. Second, we use GVM to execute *randomly generated tests* [Pacheco et al. 2007], which do not necessarily have common execution paths.

We compared our test generation algorithms to those that run on existing Java interpreters: Oracle JVM (i.e., `java` with `-Xint` option) and Java Pathfinder (JPF) [Java Pathfinder 2019; Visser et al. 2003], as well as Oracle JVM with JIT. We used several data structures from the original work on systematic sequence-based test generation and follow-up studies [Pacheco et al. 2007; Visser et al. 2003]. Additionally, GVM achieves performance parity with JVM with JIT using many CPU threads; the best configuration of GVM even outperforms JVM with JIT. Our evaluation, using

several classes from open-source projects, also shows that executing randomly generated tests on GVM outperforms sequential execution on JVM interpreter and JVM with JIT.

The main contributions of this paper include:

- ⋆ The first system, dubbed GVM, for running lightweight Java bytecode entirely on GPUs. GVM targets applications that require execution of a large number of similar tasks and share code.
- ⋆ The first work on parallel sequence-based test generation and execution of randomly generated test cases on GPUs.
- ⋆ The first set of algorithms for sequence-based testing appropriate for massively parallel environments that avoids redundant test method sequences.
- ⋆ Evaluation of GVM for two use cases: sequence-based test generation and execution of randomly generated tests. We also compare GVM with two existing interpreters and Oracle JVM with JIT to understand the current benefits and limitations. Our evaluation further compares and contrasts results of sequence-based test generation when using multiple CPU threads and multiple GPUs.

Artifacts related to GVM are available at: http://cozy.ece.utexas.edu/gvm

2 BACKGROUND

This paper adopts NVIDIA nomenclature because we use NVIDIA GPUs and CUDA [NVIDIA 2019]. The same concepts generalize to other GPU hardware and frameworks [Blythe 2006; Khronos 2019].

A CUDA program offloads computation to a GPU by calling massively multi-threaded parallel functions or *kernels*. The programming and execution models expose a tiered hierarchy of threads. At the lowest level, groups of cooperating threads are mapped to streaming multiprocessors (SM) and executed concurrently. SMs comprise an L1 data cache, scratch-pad memory, and a small number of SIMD or vector cores. GPU kernels are launched with tens or hundreds of CTAs, or *cooperative thread arrays* which over-subscribe the SMs; a hardware scheduler maps CTAs to SMs.

Global memory, which is accessible to all CTAs, buffers data between kernel dispatches and forms the basis of CPU-GPU communication. CTAs execute in SIMD chunks called *warps*, which comprise 32 threads in NVIDIA hardware. Warps are the most basic unit of scheduling on the GPU. Warps are also the primary unit of vectorized execution, so instructions in a warp are processed in lock step. As a result, GPU execution is most efficient when all threads in a warp follow the same control path. Additionally, GPUs typically comprise 10s of SMs, each with multiple vectorized execution units, each thread of which can produce unique high-latency loads or stores on any given cycle. Consequently, the demand for memory bandwidth is a critical performance concern. The hardware deals with this by hiding memory latency (a cache miss can be 100s of cycles) with aggressive hardware multi-threading, switching warp contexts frequently at instruction granularity whenever loads or stores miss in the cache hierarchy. The vectorized execution model and need to hide memory latency with memory-level parallelism are key performance determinants for GPU codes, and are characterized with first-class metrics: memory efficiency and thread divergence [Kerr et al. 2009].

Memory efficiency characterizes global memory access spatial locality and the degree to which code utilizes available bandwidth to global memory. GPUs coalesce global memory accesses for threads whenever possible, but accesses that are not sequentially aligned can result in separate transactions for each element requested, reducing memory bandwidth utilization. Conversely, higher memory efficiency translates to higher bandwidth utilization and thus performance. It is common for GPU codes to be optimized to maximize memory coalescing and by maximizing the degree to which in-flight memory references can be overlapped with arithmetic operations.

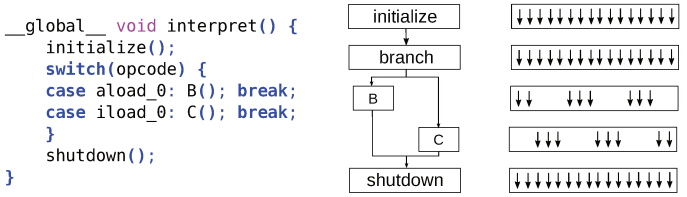


Fig. 1. Serialization of divergent control flow in a SIMD/SIMT execution model. The hardware executes both branches of the conditional, ensuring that only the effects of vector lanes corresponding to the case being executed are made visible by the hardware. The impact on performance is that control execution latency is the *sum* of the latency for each distinct control flow path.

```
typedef struct {
  handle uid; handle nameref;
  uint16_t instance_fields_count;
  uint16_t static_fields_count;
  handle fields_index;
  uint16_t methods_count;
  handle methods_index;
  handle superclass;
  handle constant_pool_index;
  // ...
} Classinfo;
```

(a) Class representation

```
typedef struct {
  handle uid;
  uint16_t max_locals;
  uint16_t num_args;
  handle declaring_class;
  handle code_index;
  handle exception_table_ix;
  int8_t native_id;
  // ...
} Methodinfo;
```

(b) Method representation

```
typedef struct {
  handle declaring_class;
  handle type;
  uint32_t offset;
  // ...
} Fieldinfo;
```

(c) Field representation

Fig. 2. Representation of a class, method, and field in GVM. Unlike existing interpreters that run on CPU, GVM has a unique layout without any pointer, non-primitive values, and data structures.

```
class BinTree { ...
  private Node root;
  void add(int x) {...}
  boolean remove(int x) {...} }
```

(a)

```
class Node { ...
  public int value;
  public Node left;
  public Node right; }
```

(b)

Fig. 3. Simplified binary search tree example in Java.

Thread divergence (see Figure 1) occurs when threads within a warp take different control paths, forcing the hardware to serially execute each branch path taken. When executing conditional code, a warp will first execute the "if" branch, and then the "else" part. The hardware deals with the effects of divergent control flow with hardware predication, effectively disabling threads that are not on the currently executing control flow path by discarding their architecturally visible changes. Non-uniform control flow can incur significant performance penalties because execution latency is the sum of the latency for each distinct control path. Consequently, codes with abundant irregular control flow and pointer-chasing can effectively under-utilize the GPU's parallel hardware.

GVM runs a large number of parallel TINYBEEs by assigning a GPU thread to each. Each TINYBEE performs a different task (e.g., by generating a different sequence of tests), so there is significant potential for high thread divergence and low memory efficiency. Because they can become the first order term for performance, improving memory efficiency and minimizing divergence are key design goals for GVM. GVM minimizes the impact of thread divergence and improves memory efficiency with a combination of data layout, algorithm design, and thread scheduling techniques, as we discuss in detail in Section 3.

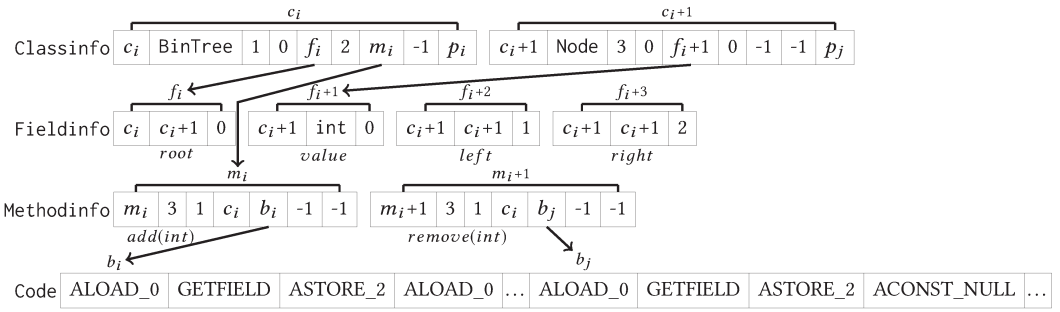


Fig. 4. Illustration of the packaged `Classinfo`, `Methodinfo`, and `Fieldinfo` for the example in Figure 3; -1 indicates that the entry should not be used or an entity does not exist.

3 GPU-BASED JAVA BYTECODE INTERPRETER

This section describes a high-level overview of the main execution phases in GVM, state representation of bytecode interpreters, and supported Java features.

3.1 Phases and Data Layout

GVM has four phases: packaging, transferring, scheduling, and interpreting.

Packaging phase: In the initial phase of the execution, which runs on the host CPU, GVM takes all classes available on the classpath and packages their content in a format suitable for interpreters running on GPUs. Our representation of classes, which is similar to the original Sun's Java, differs from existing interpreters that run on a CPU (e.g., Jikes RVM and JPF) as we assiduously avoid data structures and reference patterns that introduce additional indirection and cause unnecessary control flow or irregular memory access patterns on the GPU. GVM avoids pointers and nested structures: fields that would be pointers in a CPU-based implementation are replaced by *handles*, which turn pointer indirection into offset-based indexing into arrays. This makes the in-memory representation of data structures address-space independent, so pointers need not be updated as classfiles move between CPU and GPU memory. To maximize memory efficiency, GVM does not use any container data structures other than arrays.

We first describe the packaging of individual classfiles and then our techniques for merging packaged classfiles together. (Our description of the layout does not describe every detail of our implementation; we cover the key steps that are also needed for later text.) The design of GVM envisions the packaging to be directly implemented by the Java compiler; our prototype implements this out-of-band by post-processing classfiles produced by an unmodified Java compiler.

There is one instance of the `Classinfo` struct per classfile; a part of `Classinfo` is shown in Figure 2a. Each class has a unique `uid` (which is the handle for that class), a handle of its name on the heap, number of instance and static fields, a handle of the first field, number of methods, a handle of the first method, a handle of its parent class, and a handle of its constant pool. For each method in a class there is an instance of the `Methodinfo` struct; a part of `Methodinfo` is shown in Figure 2b. Besides a unique `uid`, there is also the number of arguments and maximum local variables, a handle of declaring class, a handle of the beginning of bytecode instructions for this method, and a handle to the exception table. Similarly, each field in the class has an instance of the `Fieldinfo` struct (Figure 2c). Each field has a handle of the declaring class, a handle of its type, and the offset in an object.

GVM uses arrays to encode bytecode instructions for all methods in the class (`int8_t[]`), exception tables for all methods in the class (`int32_t[]`), and a constant pool for the class (`int32_t[]`).

We encode `long` and `double` values with two `int32_t`. Each constant in the pool takes an extra slot that keeps the type of the constant (e.g., `CONSTANT_Integer`). Each exception table entry takes three slots: the first slot is the starting bytecode index supported by the exception handler, the second slot is the ending bytecode index supported by the exception handler, and the third slot is the bytecode index of the exception handler. All entries in the exception table are kept in the same order as in the original bytecode, so that the handlers are checked in the appropriate order once an exception is thrown.

Once each classfile is packaged, GVM performs *global packaging* of all classfiles together. Figure 4 illustrates the result of the global packaging for example classes in Figure 3; note that Figure 4 does not show classes from the Java Class Library (JCL). GVM creates one array for all instances of `Classinfo`, `Methodinfo`, `Fieldinfo`, code, exception tables, and constant pools. During the global packaging, GVM assigns unique ids (i.e., handles) to classes, methods, and fields; each unique id corresponds to the index of the element in the array. Additionally, GVM initializes all handle fields to appropriate values.

GVM currently packages classes from JCL at the beginning of the arrays to enable further optimization as those classes need not be re-packaged across runs.

Transferring phase: In the second phase, which is initiated on the host CPU, GVM allocates memory on the device and transfers the data packaged in the previous phase to the GPU. There is only one copy of each array, because the packaged data is constant and shared by all interpreters.

In the transferring phase, GVM also allocates device memory for the heap (`int32_t[]`) and static area (`int32_t[]`). Unlike shared constant data for classfiles, each TinyBee has a separate heap and static area. The static area contains one handle for each class in the system; these handles are initialized during class loading with locations of class metadata (i.e., `Class`) on the heap. A part of the heap is initialized on the host CPU to include all `String` constants from the constant pools for all classes in the system; this part of the heap will be shared among all TinyBees. The rest of the heap and the entire static area are initialized with zeros. To ensure in-memory state is address-space independent at arbitrary points in TinyBee execution, GVM uses handles for transient data rather than traditional pointers (see Section 3.2).

Scheduling phase: In the scheduling phase, GVM determines the number of TinyBees to spawn at once, as well as the way TinyBees are assigned to warps. GVM currently supports two scheduling approaches: (a) assign one TinyBee to each thread in a warp, and (b) assign one TinyBee to one warp (and leave 31 threads in the warp unused). GVM uses profile-guided heuristics to determine the appropriate assignment, i.e., each scheduling decision is based on prior runs. GVM measures thread divergence dynamically using the non-predicated warp execution efficiency (WNPEE) metric reported by `nvprof`. Conceptually, when WNPEE is below a configurable threshold (25% in our evaluation), GVM schedules only one TinyBee per warp, and otherwise defaults to scheduling one TinyBee per hardware thread. Such a design decision has not been proposed or evaluated in prior work, but we believe that it is very much worth further research.

Interpreting phase: Finally, GVM triggers the interpreting phase. Each interpreter determines its own part of the heap, the static area, and main method to execute based on the GPU thread id. Next, each interpreter dedicates local memory for the operand stack and stack frames, and initializes necessary data (e.g., base pointer). Finally, each interpreter enters the loop to execute the bytecode instructions; sections 3.2 and 4 provide more details about this phase.

3.2 State Representation and State Update

To illustrate the way we represent the program state and use handles, we give operational semantics rules for a subset of Java bytecode instructions. Our goal is to illustrate program state manipulations

$$\frac{top' = top + 1 \qquad OS' = OS[FS(bp + index + 1)/top]}{\langle \texttt{aload}\ index, pc, bp, top, FS, OS, \cdots\rangle \Rightarrow \langle CO(pc+2), pc+2, bp, top', FS, OS', \cdots\rangle}$$

$$\frac{top' = top - 1 \qquad FS' = FS[OS(top-1)/bp + index + 1]}{\langle \texttt{astore}\ index, pc, bp, top, FS, OS, \cdots\rangle \Rightarrow \langle CO(pc+2), pc+2, bp, top', FS, OS', \cdots\rangle}$$

$$\frac{OS' = OS[DA(OS(top-1)) + FIs(CP(index)).\text{offset}/top - 1]}{\langle \texttt{getfield}\ index, pc, top, OS, DA, \cdots\rangle \Rightarrow \langle CO(pc+3), pc+3, top, OS', DA, \cdots\rangle}$$

$$\frac{top' = top - 2 \qquad DA' = DA[OS(top-1)/OS(top-2) + FIs(CP(index)).\text{offset}]}{\langle \texttt{putfield}\ index, pc, top, OS, DA, \cdots\rangle \Rightarrow \langle CO(pc+3), pc+3, top', OS, DA', \cdots\rangle}$$

$$\frac{top' = top + 1 \qquad OS' = OS[SA(FIs(CP(index)).\text{declaring\_class} + FIs(CP(index)).\text{offset})/top]}{\langle \texttt{getstatic}\ index, pc, top, OS, DA, \cdots\rangle \Rightarrow \langle CO(pc+3), pc+3, top', OS, DA', \cdots\rangle}$$

$$\frac{top' = top - 1 \qquad DA' = DA[OS(top-1)/SA(FIs(CP(index)).\text{declaring\_class}) + FIs(CP(index)).\text{offset}]}{\langle \texttt{putstatic}\ index, pc, top, OS, DA, \cdots\rangle \Rightarrow \langle CO(pc+3), pc+3, top', OS, DA', \cdots\rangle}$$

$$\frac{top' = top - 1 \qquad OS' = OS[DA(OS(top-2) + OS(top-1))/top-2]}{\langle \texttt{aaload}, pc, top, OS, \cdots\rangle \Rightarrow \langle CO(pc+1), pc+1, top', OS', \cdots\rangle}$$

$$\frac{top' = top - 3 \qquad DA' = DA[OS(top-1)/OS(top-3) + OS(top-2)]}{\langle \texttt{aastore}, pc, top, OS, DA, \cdots\rangle \Rightarrow \langle CO(pc+1), pc+1, top', OS, DA', \cdots\rangle}$$

$$\frac{nxt' = nxt + CIs(CP(index)).\text{instance\_fields\_count} + 1 \qquad OS' = OS[nxt+1/top] \qquad DA' = DA[CP(index)/nxt]}{\langle \texttt{new}\ index, pc, top, nxt, OS, DA, \cdots\rangle \Rightarrow \langle CO(pc+3), pc+3, top+1, nxt', OS', DA', \cdots\rangle}$$

$$\frac{\begin{array}{c} OS' = OS[nxt+2/top-1] \qquad DA' = DA[OS(top-1)/nxt] \\ DA'' = DA'[intt\_ix/nxt+1] \qquad nxt' = OS(top-1)+2 \end{array}}{\langle \texttt{newarray}\ intt\_ix, pc, top, nxt, OS, DA, \cdots\rangle \Rightarrow \langle CO(pc+3), pc+3, top, nxt', OS', DA'', \cdots\rangle}$$

Fig. 5. Operational semantics rules for interpreters for a subset of bytecode instructions; the goal is to illustrate the way we manipulate transient data, including heap, static area, and type information (rather than to give operational semantics to all well-known Java bytecode instructions).

because our layout of the state differs from existing interpreters; our goal is *not* to change semantics of any bytecode instruction. A key insight is that our state representation relies on handle-based access to *all* program state, meaning even transient data such as the heap are accessed through offsets rather than pointers. This makes state representation address-space independent at all times, enabling individual TinyBees to be context switched at arbitrary points during program execution. While our prototype does not yet rely on this feature, we envision it to be an important feature for GVM to efficiently deal with hardware over-subscription in any production deployment.

Configuration for a TinyBee includes several components:

$$\langle\, inst, pc, sp, bp, top, nxt, FS, OS, DA, SA \,\rangle$$

where *inst* is a bytecode instruction; *pc* is the program counter; *sp* is the stack pointer for stack frames; *bp* is the base pointer for stack frames; *top* is the top of the operand stack (pointing to the first available slot); *nxt* is the next free location on the heap; *FS* is the frame stack; *OS* is the operand stack; *DA* is the heap area; *SA* is the static area. *inst* refers to all bytes in the instruction, and we use symbolic names for the opcode and other bytes in our rules. All non-primitive components are

implemented as `int` arrays; *FS*, *OS*, *DA*, *SA* are partial functions from array indexes to integers ($Int \rightharpoonup Int$). Additionally, we assume that code (*CO*), constant pool (*CP*), `Fieldinfos` (*FIs*), `Methodinfos` (*MIs*), and `Classinfos` (*CIs*) are available in the context. As discussed earlier these are constant throughout the execution; we use a partial function from array indexes to integers for code ($Int \rightharpoonup Int$), and partial functions from array indexes to an instance of a `Fieldinfo`, `Methodinfo`, and `Classinfo` for *FIs*, *MIs*, and *CIs*, respectively. Note that *inst* is equal to $CO(pc)$, but we carry both for convenience to avoid parsing bytes of each instruction within our rules.

Figure 5 shows the rules for TinyBees. We cover a couple of representative groups, e.g., local variable read/write, field read/write, array read/write, and allocation. In each rule, we show only the relevant part of the configuration, i.e., components that are accessed or modified. We use $\cdots$ to denote other components in the configuration [Ellison and Roşu 2012]. We also use the following operations: (1) component lookup _(_) ($Component \times Int \rightharpoonup Int$), (2) component update _[val/loc] ($Component \times Int \times Int \rightarrow Component$), and (3) component member access _._ ($Component \times Name \rightharpoonup Int$).

For the simplicity of exposition, the rules do not show class loading, exception handling, and assume that applications only use (small) integer constants and variables. Our implementation has none of these limitations. We use the following stack frame and the frame stack grows towards larger memory addresses:

...	higher address
locals (local_0 is at the lowest address)	
old base pointer	
current_method handle	
top of the operand stack	
return address/pc	↑
...	lower address

Examples: We briefly describe the rules for several instructions. `aload` is a two-byte instruction that loads a reference onto the operand stack from a local variable *index* (which is given as the second byte). The local variable is obtained from the frame stack $FS(bp + index + 1)$. The value is placed on top of the operand stack and the stack pointer is moved to the next location.

`getfield` is a three-byte instruction that gets a field value of an object; the second and third bytes specify index of the field being accessed. The rule first finds the offset of the field $FIs(CP(index))$.offset and then finds the value of the field on the heap. The obtained value is placed on the operand stack; we do not update the top of the stack because the rule overrides the object reference with the fetched value.

The `new` instruction, which allocates an object on the heap, has three bytes; the first byte is the opcode and the remaining two bytes constitute the index into the constant pool that contains `Classinfo` corresponding to the type to be allocated. At the next available place on the heap we put the handle for the type of the object ($DA' = DA[CP(index)/nxt]$). On top of the operand stack we place the handle for the object, which is the first location after the type ($OS' = OS[nxt + 1/top]$). The rule also updates the top of the operand stack, as well as the next available location on the heap; the next location is immediately after the fields of the allocated object.

Finally, `newarray` instruction allocates an array of the given type; the type is given as the second byte and it is an index into the constant pool for the current class. On the heap, we first place the length of the array ($DA' = DA[OS(top - 1)/nxt]$); the length for `newarray` is given on top of the operand stack. Next, we place the type of the elements at the next location on the heap ($DA'' = DA'[intt\_ix/nxt + 1]$). On top of the operand stack we place the array handle, i.e., index of the first element after the element type. As the final step, the rule updates the pointer to the next available location on the heap.

4 IMPLEMENTATION DECISIONS

This section briefly describes our implementation decisions and touches on several limitations; Section 8 describes the limitations in much greater detail.

Bytecode instructions: GVM supports the Java 7 language specification and class-format version 51. Thus, it supports all existing Java bytecode instructions except `invokedynamic`, `monitorenter`, and `monitorexit`. `invokedynamic` simply requires additional engineering. As our current focus is on sequential applications, we had no need to support `monitorenter` and `monitorexit`; supporting concurrency in GVM is an exciting future direction.

Java Class Library (JCL): Code packaged for the execution on GPUs includes many classes from JCL (mostly from `java.lang` and `java.util` packages). The list of classes to be included is configurable and specified manually at the moment. Recall that GVM does not require any change to the original source code for any class.

Native methods: Several methods in JCL are not written in Java, i.e., these methods are implemented as native method. Clearly, these methods cannot be directly executed by a bytecode interpreter [Visser et al. 2003]. If native methods have side effects, we have to implement those methods to ensure that the effect is reflected in the memory of the interpreter. We support native methods via an interface similar to JNI [Oracle 2019c]. So far, we have written in CUDA over 90 native methods that we encountered in JCL (e.g., `Object.clone`); JCL has around 900 native methods, but not all native methods make sense to run on GPU.

Reflection: We used our support for native methods to implement various methods related to reflection API in Java. We support methods in `java.lang.reflect.Array`, `java.lang.Class`, etc. Our support for reflection so far was demand driven.

Garbage collector (GC): tinyBee includes no garbage collector. Specifically, we follow what the Java community calls: Epsilon Garbage Collector [Oracle 2019b], i.e., a passive GC implementation with bounded allocation limit that crashes the application once the memory limits are exceeded. This approach is described in JEP 318 [Oracle 2019b], and support for the EpsilonGC was officially released in Java 11. As described in the JEP, this GC is ideal for "Extremely short lived jobs". As we will see in the next section, we can implement support for proper GC algorithms because we already have full heap traversal support.

GPU memory: We keep most of the data in global memory. We do not utilize the constant memory, because the size of the packaged code far exceeds the limits for the constant memory; we also did not use texture memory as it provided no performance improvements based on our initial profiling.

Exception handling: Our support for exception handling is straightforward. Once an exception is thrown, we find the handle for the current method; recall (Section 3.2) that we keep this info readily available in each stack frame. Using the method handle we access the exception table and check if any exception handler for the current method can catch the thrown exception. If yes, we extract the next value for the pc from the exception table; otherwise, we find the previous method on the frame stack and repeat the steps. Finally, if no exception handler is found, the tinyBee halts the execution and sets an appropriate exit flag.

5 SEQUENCE-BASED TEST GENERATION

This section presents the first use case of GVM. We describe systematic sequence-based test generation [Visser et al. 2006], as well as our algorithms for parallel sequence-based test generation on GPUs by utilizing GVM. Our algorithms are the first effort to parallelize sequence-based test generation and they are also applicable to parallel runs on CPUs.

```
class BinTreeDriver {
  public static void main(String[] args) {
    // int M = ...; int N = ...; int P = 1;
    BinTree t = new BinTree();
    for (int i = 0; i < M; i++) {
      int v = VS_toss(N);
      int p = VS_toss(P);
      switch (p) {
        case 0: t.add(v); break;
        case 1: t.remove(v); break; }
      ignoreIf(store(cksum(t))); }}}
```

Fig. 6. Test driver in Java for the BinTree example.

5.1 Test Generation Drivers

Sequence-based test generation techniques have been used for testing Java container classes [Pacheco et al. 2007; Visser et al. 2006]. As the input, these techniques take a system under test, list of methods of interest, and max length of each sequence. For the given input, these techniques exercise *all* method sequences up to the given sequence length. For example, for the `BinTree` class in Figure 3a, the following list includes all the sequences of length two:

`t.add(val); t.add(val);` // $\boxed{val} \xrightarrow{\text{right}} \boxed{val}$
`t.add(val); t.remove(val);` // empty tree
`t.remove(val); t.add(val);` // $\boxed{val}$
`t.remove(val); t.remove(val);` // empty tree

The original work on systematic sequence-based test generation was built on top of the Java Pathfinder (JPF) software model checker [Java Pathfinder 2019; Visser et al. 2003]. JPF implements a backtrackable Java bytecode interpreter; JPF itself is implemented in Java and it executes programs sequentially. To check all sequences of method calls, a user has to write a *test driver* as shown in Figure 6. The driver creates an instance of a tree and iterates over the loop `M` times; `M` is the maximum length of method sequences. In each iteration of the loop, the driver chooses a value between [0, N), one of the operations (`add` or `remove`) and invokes the operation with the selected value on the tree instance. The driver uses the `VS_toss(n)` [Godefroid 1997] function, which forces the current execution to pick one value [0, N) and schedule another execution that will enforce the subsequent value (until all values are exhausted). We discuss the code that invokes `ignoreIf` in Section 5.2. An implementation of the test driver in Java (without `VS_toss` and software model checker that explores all values) requires several nested loops [Sharma et al. 2011].

We next describe two test generation *modes* that use the driver in Figure 6; for each mode we present our parallel algorithm for test generation on top of GVM. Our goal was to design algorithms to ensure that the executed sequences exactly match those sequences from prior work. In other words, our goal was not to increase the quality of generated sequences but to accelerate generation of sequences that were already evaluated in prior work.

5.2 Shape Matching Mode

Shape matching was introduced to avoid the exponential explosion of exploring all sequences up to the given length [Visser et al. 2003]. The key idea is to avoid extending method sequences from a state that has been seen before. For example the following two short method sequences would lead to the same shape: `t.remove(0)` and `t.remove(1)`. In both cases the resulting tree instance would be empty, and only one of those sequences should be extended further.

Require: M - sequence length
Require: N - number of values
Require: P - number of operations
1: **function** HOST()
2: $inQueue \leftarrow$ INIT(N, P)
3: $cksum2Tid \leftarrow$ EMPTYMAP(C)
4: **for all** $level \in 1$ *to* M **do**
5: **for all** $iter \in 0..(\lceil total/THRS \rceil - 1)$ **do**
6: ENGINE$\langle\langle\langle$THRS$\rangle\rangle\rangle(iter * THRS, inQueue, cksum2Tid)$
7: **end for**
8: GENNEXT$\langle\langle\langle$C$\rangle\rangle\rangle(inQueue, cksum2Tid)$
9: $size \leftarrow inQueue.size$
10: SORT$\langle\langle\langle$SIZE$\rangle\rangle\rangle(inQueue)$
11: **end for**
12: **end function**
13: **function** ENGINE$(base, inQueue, cksum2Tid)$
14: $jid \leftarrow base + blockDim.x \times blockIdx.x + threadIdx.x$
15: $choices \leftarrow inQueue[jid]$
16: INTERPRET$(choices)$
17: $cksum \leftarrow$ SHAPECHECKSUM()
18: ATOMICMIN$(cksum2Tid[cksum], jid)$
19: **end function**
20: **function** GENNEXT$(inQueue, cksum2Tid)$
21: $jid \leftarrow blockDim.x \times blockIdx.x + threadIdx.x$
22: $inId \leftarrow cksum2Tid[jid]$
23: **if** $inId = \perp \ \vee \ inId < 0$ **then**
24: **return**
25: **end if**
26: $cksum2Tid[jid] \leftarrow -inId - 1$
27: $choices \leftarrow$ ATOMICREMOVE$(inQueue, inId)$
28: ADDALLATOMIC$(inQueue,$ EXTEND$(choices, N, P))$
29: **end function**

Fig. 7. Algorithm for the shape matching mode on GVM. We use the standard notation KERNEL<<<THRS>>> to invoke a kernel on a GPU and spawn THRS threads in parallel. We also use CUDA built-in variables [NVIDIA 2019] (`blockDim.x`, `blockIdx.x`, `threadIdx.x`) to assign a unique id to each GPU thread.

In this mode, the test driver invokes `ignoreIf(store(cksum(t)))` (Figure 6). The `cksum` method takes as input a root of an object graph on the heap and returns checksum of the *shape of the object graph* without values for primitive fields. The implementation simply traverses the object graph in depth-first style and computes the checksum of the shape. The `store` method returns `true` if the shape has been seen before; `false` otherwise. Finally, the `ignoreIf` method stops the execution of the current sequence if the argument evaluates to `true`.

Prior work on shape matching showed that *breadth-first* execution of method sequences outperforms other traversal strategies in terms of code coverage for code under test. In the breadth-first order all sequences of length 1 are executed before sequences of length 2.

Figure 7 shows *our novel algorithm* for systematic execution of method sequences with shape matching on top of GVM. We designed the algorithm to guarantee that the same sequences of method calls are executed as with the sequential breadth-first traversal. This is also the first parallel algorithm for sequence-based test generation with shape matching.

The inputs to the algorithm include: sequence length (M), number of distinct values to be used as arguments to operations (N), and the number of operations (P). In the host code on CPU we

allocate two data structures; both structures are allocated in the device memory. First, we allocate a queue (`inQueue`) that will keep choices to be executed for the next level (in breadth-first order). This queue is initialized with all pairs for N and P; for $N = 2$ and $P = 2$, the initial queue would contain: $(0, 0), (0, 1), (1, 0), (1, 1)$. Second, we allocate an array of integers (`cksum2Tid`), with length `C`, that represents a map from the state checksum (index of the array) to the thread id (value in the array) of the thread that executed a sequence of operations that resulted in that checksum.

The `genNext` function prepares choices for the next level of the algorithm. Specifically, there are as many hardware threads running `genNext` as potential checksums (`C`). Each thread takes a checksum value, based on the thread id, and returns immediately if the value is equal to the sentinel value or the value is negative. We use negative values to encode choices that were extended on one of the previous levels. If a thread finds that the checksum was first seen on the current level, then it sets the value in the checksum map to negative (for future invocations of this kernel) and then extends choices for the next level based on N and P.

We give an intuition that our algorithm executes the same method sequences as the sequential algorithm. In our algorithm, the outer loop on the host enforces that sequences of length k are executed before sequences of length $k + 1$. The algorithm sorts `choices` to ensure that a thread with smaller id executes a sequences that is lexicographically smaller (in terms of values in `choices`). If two threads execute sequences that lead to the same state checksum, our algorithm keeps the thread with smaller id and thus an earlier sequence in lexicographical order. Using a negative value in `cksum2Tid` ensures that we do not extend the same sequence twice and also ensures that any longer sequence that result in previously seen checksum is ignored.

It is important to mention that our algorithm also has a property to schedule two choices that differ only in the last level in two neighboring threads; this decreases divergence (Section 2), and consequently, improves performance.

5.3 Complete Matching Mode

Shape matching may end up being too aggressive in reducing the number of executed sequences. *Complete matching* uses both shape and primitive values to compute the checksum of a program state [Visser et al. 2003]. Thus, test drivers for complete matching remain almost the same as for shape matching; the only difference is that `cksum` invocation in this case computes complete matching. Similarly, our algorithm in Figure 7 remains unchanged; we only change the function that computes the checksum of the program state.

5.4 Multithreaded CPU Algorithm

We have adapted our algorithm from Figure 7 to CPUs. Our implementation is written in pure Java and uses the `Thread` class. Specifically, instead of invoking the `engine` function (line 6), we start N CPU threads, where N is specified by a user, and each thread processes an equal number of input sequences. We follow a similar approach and spawn new CPU threads when invoking `genNext` (line 8). To sort the input queue prior to the next level (line 10), we use `Collections.sort`. We have also experimented with the `Executors` and `ExecutorService` classes, but discovered that those implementations were much slower than our current implementation with `Threads`.

5.5 Multi-GPU Algorithm

We have also adapted our algorithm to enable sequence-based test generation on several GPUs simultaneously. Most parts of the algorithm from Figure 7 remain unchanged. The key difference is that we need to split the input evenly at each level and communicate among GPUs after all `engine` calls are finished at each level (line 7) to ensure that there are no duplicate shapes generated on different devices. Specifically, each device broadcasts its `cksum2Tid` to all other devices. Then each

device finds a minimum value for every index across all `cksum2Tids` and keeps the minimum value in its local `cksum2Tid`. After this step, all GPU devices have the same view of the checksum table. We invoke `genNext` on a single GPU device; execution of the `genNext` function is much faster than execution of the `engine` function, so we do not utilize multiple GPUs for this task.

6 EXECUTION OF RANDOMLY GENERATED TESTS

Feedback-directed random test generation, as implemented in Randoop [Pacheco et al. 2007], was introduced as an alternative to systematic sequence-based test generation. Rather than executing all method sequences systematically, Randoop randomly chooses a method to invoke and arguments for the method from the argument pool. Initially, the argument pool contains a small number of constants, e.g., `0`, `-1`, `1`, `null`. If a method invocation is successful (e.g., no exception is thrown), Randoop saves the method sequence, which can be extended in the future if the result of the method sequence can be used as an argument for another randomly chosen method. Randoop generates a large number of short-running tests, and each test includes some test oracles. Clearly, these tests share the same codebase. Thus, we expect that GVM will accelerate the execution of generated tests. At the same time, we expect high divergence, as there are no guarantees that any two sequences of method calls will be similar. Our scheduling plays an important role to detect high divergence and schedule a single tinyBee in one warp.

7 EVALUATION

This section describes our evaluation of GVM. We show our research questions, describe hardware and software configurations, introduce subjects under study, describe runtime environments, show the results of our experiments, answer the questions, and discuss some design decisions.

7.1 Research Questions

To evaluate the current state of GVM and our parallel algorithms for systematic sequence-based test generation, we answer the following research questions:

- **RQ1:** How does GVM compare to sequential execution on the existing bytecode interpreters and JVM with JIT for performing sequence-based test generation?
- **RQ2:** What is the performance improvement obtained on GVM by manually transforming ("jitting") Java bytecode to native CUDA code?
- **RQ3:** What are the speedups obtained by parallelizing sequence-based test generation with multiple CPU threads and GPUs?
- **RQ4:** What is the impact of the number of tinyBees on GVM's performance?
- **RQ5:** How good is the performance of GVM when executing randomly generated tests?
- **RQ6:** How do the thread divergence and memory efficiency impact GVM in various scenarios?

7.2 Hardware and Software Configuration

We ran all the experiments on a system with two Intel(R) Xeon(R) 12-core CPUs E5-2650 v4 @ 2.20GHz with 128GB of RAM, running Ubuntu Linux 18.04 LTS. We used Oracle Java 1.8.0_181. For the experiments run on GPUs, we used (up to four) Tesla P100-SXM2 and CUDA version 10.0.

7.3 Subjects

We evaluated test generation algorithms using six data structures used in prior studies on test input generation [Kuraj et al. 2015; Pacheco et al. 2007; Visser et al. 2006]. The first column in Table 1 shows the list of data structures. We obtained the implementation of these data structures from

publicly available repositories. Following prior work, we use two operations ($P = 2$) and we set the number of unique values (N) to be one smaller than the max sequence length (M). Our experiments did not require changes to the existing code of any subject.

To evaluate the benefits in using GVM to accelerate execution of randomly generated tests, we chose four projects and randomly selected several classes in each project. Projects include: ds - data structures used in our systematic test generation study; jcl - JCL classes; vectorz - a library for fast vector and matrix manipulation; and apache - Apache's extension of JCL. We generated 1,000 tests for each chosen class using the default Randoop configuration.

7.4 Runtime Environments

Java Pathfinder (JPF): The original work on systematic sequence-based test generation was done on top of JPF, which is the main reason that we include JPF in our evaluation. We do not use JPF for our second use case because we expect that JPF is much slower than Oracle's Java with JIT, which we used in the evaluation. JPF itself is implemented in Java and runs on a host JVM. JPF does not implement any just-in-time compilation, but JPF code gets optimized by the host JVM. We dedicate 16GB of main memory to JPF for all runs.

Oracle Java in the interpreter mode (IJVM): We use the standard Java implementation by Oracle [Oracle 2019a] in the interpreter mode, which can be enabled via `-Xint` command line option. We included this runtime environment to provide a comparison between an interpreter running on CPU vs. GPU. For the first use case, we had to rewrite test drivers to use loops instead of the `VS_toss` function. Furthermore, we use Java reflection to compute a state checksum. We dedicate 16GB of the main memory to each run of IJVM.

Oracle Java in the default mode (DJVM): We use the standard Java runtime with the default configuration. This is a highly engineered system with just-in-time (JIT) compilation. We use the same test drivers as for IJVM. We note that invoking a sequence of method calls on a few classes under test is very close to an ideal scenario for JIT. We dedicate 16GB of the main memory to each run of DJVM. The state checksum is computed the same way as for IJVM.

GVM: For the first use case we run GVM with 24,576 TINYBEES. For the second use case, we run as many TINYBEES as there are tests. For both use cases, we split the available memory evenly among the TINYBEES, with heap and stack frames consuming 48KB and 1KB per TINYBEE, respectively. In later sections, we explore the impact of several configurations (e.g., different number of GPU threads) on the performance of the system. In all experiments, we used the same test drivers as those used by JPF. We compute a state checksum via reflection, which is the same approach used for IJVM and DJVM.

7.5 GVM vs. Sequential Execution on a CPU for Sequence-Based Test Generation

RQ1: How does GVM compare to sequential execution on the existing bytecode interpreters and JVM with JIT for performing sequence-based test generation?

In this section we contrast GVM with the existing *sequential* sequence-based test generation algorithm executing on CPU on existing runtime systems. (Section 7.7 evaluates our parallel algorithms using many CPU threads.) We obtained the results for both shape matching and complete matching modes. Reported values are averages over two runs.

Tables 1 and 2 show the results for shape matching and complete matching, respectively. The first and second columns show the name of a data structure and the max sequence length, respectively. We increase the max sequence length as long as GVM, in the default configuration, has enough memory. The third column shows the total number of executed sequences; this number of executed sequences is the same for all runtime environments. Columns 4-7 show execution time

Table 1. Results for the Shape Matching Mode. SeqLen - Max Sequence Length; #Sequences - Total Number of Executed Sequences; JPF, DJVM, IJVM, GVM - Time (in Milliseconds) to Execute All Sequences With Corresponding Runtime Environment.

Structure	SeqLen	#Sequences	Execution Time [ms]				$\frac{\text{JPF}}{\text{GVM}}$	$\frac{\text{DJVM}}{\text{GVM}}$	$\frac{\text{IJVM}}{\text{GVM}}$
			JPF	DJVM	IJVM	GVM			
IntAVLTreeMap	30	18303	76067	16885	792354	11023	6.89	1.52	71.88
IntAVLTreeMap	31	23362	111996	22457	1085890	14323	7.81	1.56	75.81
IntAVLTreeMap	32	30011	139644	29888	1476901	17236	8.09	1.73	85.69
IntRedBlackTree	27	31572	103267	19572	902297	9955	10.37	1.96	90.63
IntRedBlackTree	28	46814	147096	30388	1469487	15508	9.48	1.96	94.75
IntRedBlackTree	29	69477	244311	48766	2345175	23804	10.25	2.04	98.52
BinomialHeap	63	64	2135	1019	8351	1817	1.17	0.55	4.59
BinomialHeap	64	65	2260	1065	8594	1879	1.20	0.56	4.57
BinomialHeap	65	66	2178	1061	9330	1981	1.10	0.53	4.71
BinTree	73	74	2172	1002	9199	3586	0.60	0.28	2.56
BinTree	74	75	2209	1040	9201	3709	0.59	0.28	2.48
BinTree	75	76	2395	1051	9816	3836	0.62	0.27	2.55
FibHeap	38	669	6201	2095	44630	1535	4.03	1.35	29.06
FibHeap	39	706	7335	2238	49312	1604	4.57	1.39	30.73
FibHeap	40	744	7641	2423	54460	1662	4.59	1.46	32.76
TreeMap	28	38867	91294	21356	956582	7421	12.29	2.87	128.90
TreeMap	29	55043	125521	31826	1447260	11046	11.36	2.88	131.01
TreeMap	30	77232	219167	48002	2205486	16099	13.61	2.98	136.99

in milliseconds for JPF, DJVM, IJVM, and GVM. Time for GVM is computed as the total time for transferring, scheduling, and interpreting.

7.5.1 Shape Matching Mode. Table 1 shows results for the shape matching mode. It is important to note is that for three data structures (BinomialHeap, BinTree, and FibHeap) the number of executed sequences is *very small*. This happens due to aggressive abstraction (Section 5.2). For example, for BinomialHeap and BinTree the number of method sequences is always one larger than the max sequence length. Namely, each generated instance of the tree is a list of nodes. In these cases, measuring time or using an algorithm for parallel generation is unnecessary because using a single thread for each sequence length is sufficient. Therefore, in the next paragraph, we only discuss the results for other data structures.

GVM outperforms JPF with the speedup between 6.89× (for IntAVLTreeMap) and 13.61× (for TreeMap). We consider this speedup substantial [Lee et al. 2010]. We observed that some overhead for GVM comes from extra steps for extending and sorting sequences and for state checksum computation. Due to the limited number of instances to explore, GVM provides moderate speedup over DJVM and commonly performs better for longer sequences. IJVM is much worse in this case because the state matching computation, as for DJVM and GVM, uses reflection to traverse objects on the heap and that traversal does not get optimized at runtime. We observed speedup even for

Table 2. Results for the Complete Matching Mode. SeqLen - Max Sequence Length; #Sequences - Total Number of Executed Sequences; JPF, DJVM, IJVM, GVM - Time (in Milliseconds) to Execute All Sequences With Corresponding Runtime Environment.

Structure	SeqLen	#Sequences	Execution Time [ms]				$\frac{\text{JPF}}{\text{GVM}}$	$\frac{\text{DJVM}}{\text{GVM}}$	$\frac{\text{IJVM}}{\text{GVM}}$
			JPF	DJVM	IJVM	GVM			
IntAVLTreeMap	11	18134	32211	4578	161777	1193	26.98	3.83	135.55
IntAVLTreeMap	12	45286	81467	11149	470214	2992	27.22	3.72	157.15
IntAVLTreeMap	13	105706	198529	29403	1276740	7484	26.52	3.92	170.59
IntRedBlackTree	10	12179	20051	2898	88964	1049	19.11	2.76	84.81
IntRedBlackTree	11	31596	48373	7182	265870	2728	17.73	2.62	97.45
IntRedBlackTree	12	81210	127099	18435	784316	7022	18.09	2.62	111.69
BinomialHeap	9	43048	26956	4305	138795	1172	22.98	3.67	118.37
BinomialHeap	10	90634	72891	10714	394342	2863	25.45	3.74	137.73
BinomialHeap	11	180927	162120	23360	947297	6280	25.80	3.71	150.82
BinTree	7	6265	3531	687	6830	203	17.39	3.38	33.64
BinTree	8	26416	10472	1696	35703	572	18.28	2.96	62.36
BinTree	9	90405	37731	4939	160491	1944	19.39	2.54	82.53
FibHeap	7	9420	4821	926	12363	233	20.65	3.96	52.95
FibHeap	8	37289	16097	2432	65883	549	29.29	4.42	119.89
FibHeap	9	129491	55672	8662	303967	2062	26.98	4.20	147.38
TreeMap	12	57057	60285	10586	427012	3424	17.60	3.08	124.69
TreeMap	13	121847	138101	26750	1089620	8057	17.13	3.31	135.22
TreeMap	14	241897	294816	59014	2530720	17829	16.53	3.30	141.94

cases that do not benefit from parallel execution; overall, GVM outperforms IJVM with the speedup between 71.88× and 136.99×.

7.5.2 Complete Matching Mode. Table 2 shows the results for the complete matching mode. As for the prior mode, we keep the same format for the table, so we do not discuss the details of each column. We can observe that the speedup of GVM over JPF is between 16.53× (for TreeMap sequence length 14) and 29.29× (for FibHeap sequence length 8). The speedup is larger than for the shape matching mode, because there are more sequences being executed. Similarly, GVM provides larger speedup over DJVM due to the larger number of sequences. As in the previous mode, IJVM pays a high cost to compute the state checksum.

This section considered only sequential execution on a CPU for two reasons. First, we wanted to establish an initial position of GVM with respect to the sequential execution. Second, prior work has developed only sequential algorithm for sequence-based test generation. In Section 7.7 we study parallel algorithm with multiple CPU threads and GPUs.

7.6 Manual AOT Transformation

RQ2: What is the performance improvement obtained on GVM by manually transforming ("jitting") Java bytecode to native CUDA code?

Table 3. Results for the Shape Matching Mode When Examples are Compiled Ahead of Time to CUDA.

Structure	SeqLen	#Sequences	Execution Time [ms]				$\frac{JPF}{GVM}$	$\frac{DJVM}{GVM}$	$\frac{IJVM}{GVM}$
			JPF	DJVM	IJVM	GVM			
IntAVLTreeMapAOT	30	18303	58335	15448	725162	2537	22.98	6.08	285.83
IntAVLTreeMapAOT	31	23362	89848	20848	976239	3212	27.96	6.48	303.93
IntAVLTreeMapAOT	32	30011	110569	29349	1341285	4009	27.57	7.31	334.52
IntRedBlackTreeAOT	27	31572	71671	19145	830741	2884	24.84	6.63	288.05
IntRedBlackTreeAOT	28	46814	113772	29758	1355624	4392	25.89	6.77	308.65
IntRedBlackTreeAOT	29	69477	178268	46459	2125643	6544	27.23	7.09	324.79
TreeMapAOT	28	38867	74637	21448	947177	3723	20.13	5.79	255.53
TreeMapAOT	29	55043	118089	33833	1469329	5406	21.94	6.29	273.00
TreeMapAOT	30	77232	216112	46303	2188203	7793	27.86	5.98	282.15

Table 4. Results for the Complete Matching Mode When Examples are Compiled Ahead of Time to CUDA.

Structure	SeqLen	#Sequences	Execution Time [ms]				$\frac{JPF}{GVM}$	$\frac{DJVM}{GVM}$	$\frac{IJVM}{GVM}$
			JPF	DJVM	IJVM	GVM			
IntAVLTreeMapAOT	11	18134	26610	4430	164321	845	31.48	5.24	194.46
IntAVLTreeMapAOT	12	45286	60057	11119	472365	2048	29.32	5.42	230.64
IntAVLTreeMapAOT	13	105706	160853	27844	1258355	5021	32.03	5.54	250.61
IntRedBlackTreeAOT	10	12179	13219	2909	86460	555	23.79	5.23	155.64
IntRedBlackTreeAOT	11	31596	35322	6796	262250	1366	25.84	4.96	191.91
IntRedBlackTreeAOT	12	81210	99031	18645	753733	3382	29.27	5.51	222.86
TreeMapAOT	12	57057	51471	10584	440033	2260	22.77	4.68	194.67
TreeMapAOT	13	121847	116448	26682	1071991	5362	21.76	4.98	200.38
TreeMapAOT	14	241897	261153	60253	2542028	10970	23.80	5.48	231.72

We studied the impact of ahead-of-time (AOT) compilation on our results. We performed all transformations *manually* at this point. Our approach was to convert each method in Java to equivalent code in CUDA, which still manipulates memory of the bytecode interpreters. We also simplify types by changing boxed types (`Integer`) with primitive types (`int`) in Java code. The basic motivation was to reduce the number of interpreted instructions, although memory manipulations remain similar. We also expected that this change of the `Integer` type to `int` might have negligible impact on DJVM runs.

We performed our changes on IntAVLTreeMap, IntRedBlackTree, and TreeMap and obtained IntAVLTreeMapAOT, IntRedBlackTreeAOT, and TreeMapAOT; we skipped transforming other data structures as the number of sequences was small for the shape matching mode and parallel runs are less meaningful (Section 7.5.1). Tables 3 and 4 show the results for IntAVLTreeMapAOT, IntRedBlackTreeAOT, and TreeMapAOT. The impact of AOT is clearly substantial on all interpreters. As expected, time for DJVM did not change much because DJVM optimizes code with JIT and performs more aggressive optimizations than those we did manually. We can see that the benefit of GVM is larger for the optimized code, which provides motivation to study AOT and JIT for GVM

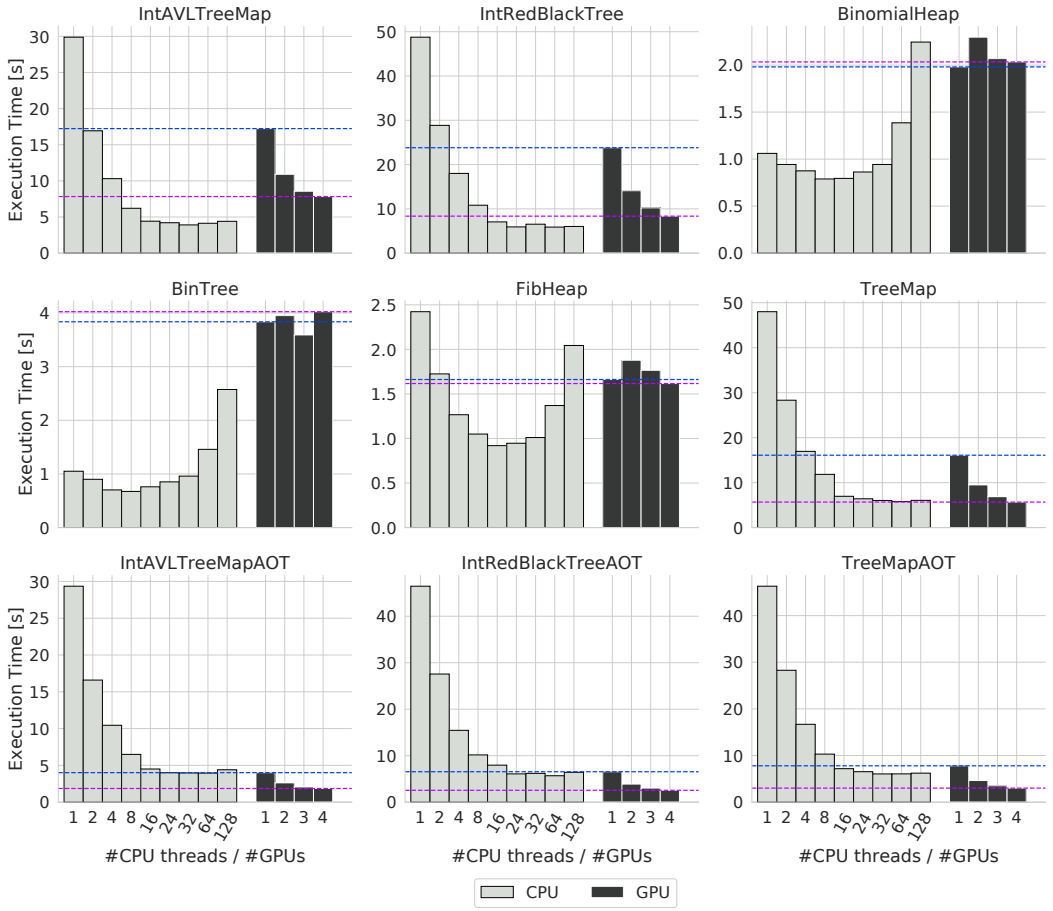


Fig. 8. Comparison of parallel generation in shape matching mode using multiple CPU threads and GPUs.

in more detail in the future. For example, if we compare results in Tables 2 and 4, we observe that ratios DJVM/GVM for TreeMap and TreeMapAOT are 2.98× and 5.98×, respectively.

7.7 Parallel Generation using Multiple CPU Threads and GPUs

RQ3: What are the speedups obtained by parallelizing sequence-based test generation with multiple CPU threads and GPUs?

Considering that there is only a sequential implementation of JPF, and GVM outperformed IJVM by a large margin, we only study performance of our parallel algorithm on DJVM. Additionally, we consider performance of the parallel algorithm if we utilize multiple GPUs. In sections 5.4 and 5.5, we have already discussed the way we adjust our parallel algorithm to multiple CPU threads and multiple GPUs. Barcharts in Figure 8 and 9 show the results of various number of CPU threads and various numbers of GPUs, with the largest max sequence length for each subject. To help with the side-by-side comparison we also draw two horizontal lines that correspond to runs on 1-GPU and 4-GPUs. For the sake of completeness, we show the results for all subjects, although the results for BinomialHeap, BinTree, and FibHeap in the shape matching mode should be ignored due to the nature of these subjects.

We can observe that the results using a single GPU frequently match the results obtain when running between 2 and 16 CPU threads. Runs for "jitted" structures are similar to the best results

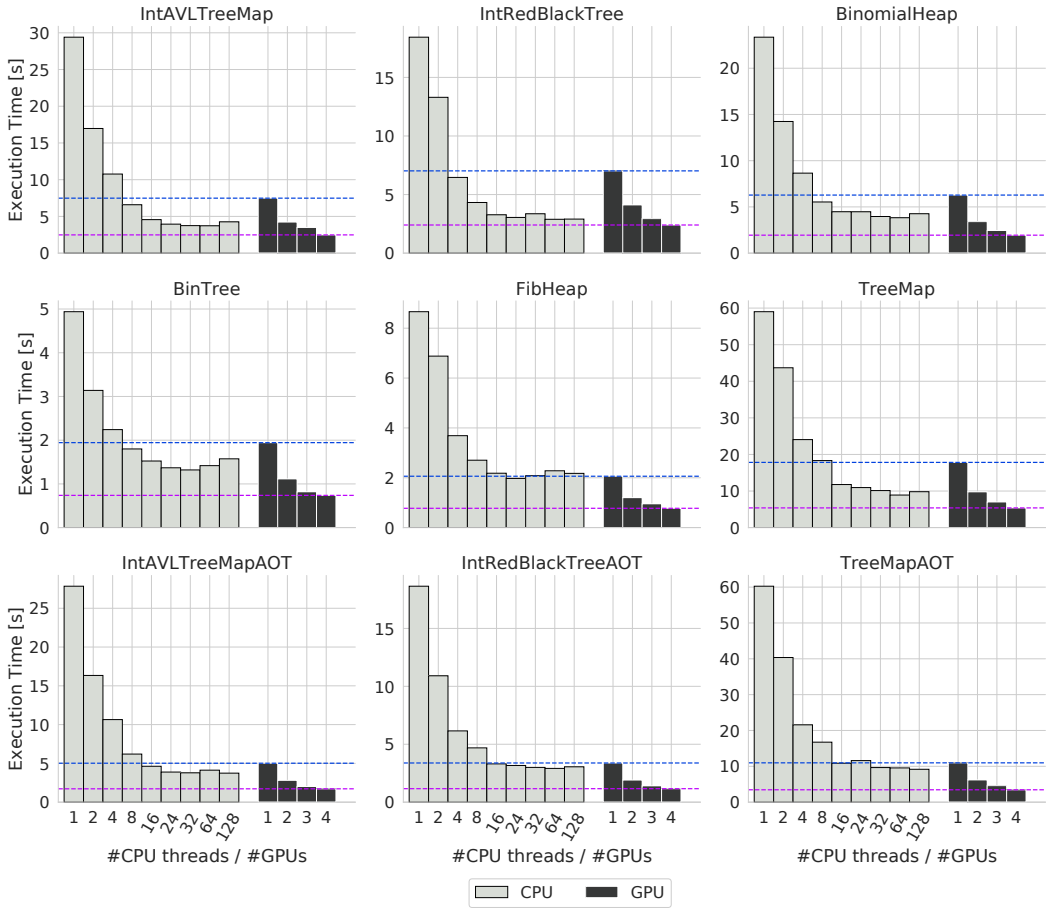


Fig. 9. Comparison of parallel generation in complete matching mode using multiple CPU threads and GPUs.

obtained on CPUs. Additionally, we can observe that runs with more than 24 or 32 CPU threads provide no extra performance benefits. Doubling the number of GPUs brings significant performance boost. Finally, GVM runs with four GPUs provide the best overall performance.

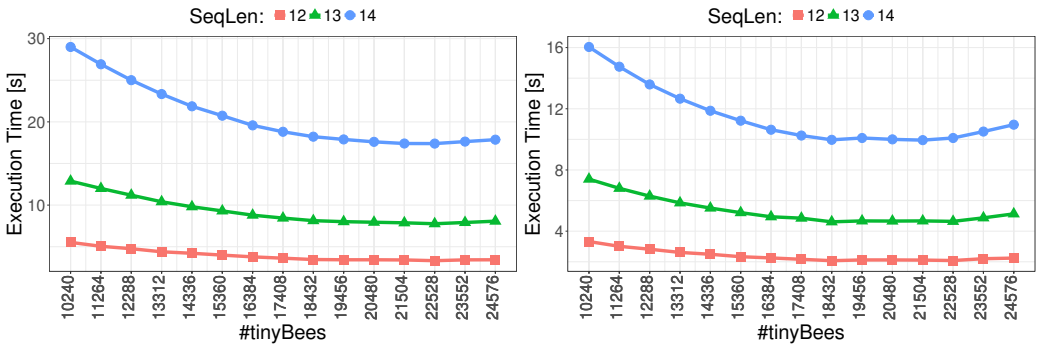
In conclusion, we find that our parallel algorithm for sequence-based test generation enables performance improvements on parallel hardware. Moreover, our results demonstrate that for a specific Java application, GVM can be as efficient as (or even outperform) DJVM. This means that GVM can be used to increase throughput on a system by running applications on both CPUs and GPUs, and further research can be done to combine DJVM and GVM.

7.8 Case Study: Number of TinyBees

RQ4: What is the impact of the number of TinyBees on GVM's performance?

We evaluated the impact of the number of TinyBees running in parallel on the performance of sequence-based test generation. In this case study, we used TreeMap and TreeMapAOT data structures and complete matching. We chose TreeMap because this structure is a widely used code from JCL. We chose complete matching because it is algorithmically challenging and does not suffer from shortcomings of shape matching.

Figure 10 show the impact of the number of TinyBees (x-axis) on total execution time (y-axis). We show the results for different max sequence lengths, which results in three different lines on



(a) TreeMap (b) TreeMapAOT

Fig. 10. Scaling GVM with various numbers (in increments of 1024) of tinyBees for TreeMap and complete matching. We show the results for three different max sequence lengths.

Table 5. Execution Time for Randomly Generated Tests.

Project	#Tests	Execution Time [ms]			
		DJVM	IJVM	GVM	GVM-Warp
ds	6000	281	276	242	121
jcl	6000	427	451	719	236
vectorz	8000	638	742	627	337
apache	8000	627	798	888	587

the plot. We can see that the increase in the number of tinyBees has a positive impact on the performance. Clearly, we can observe that the increase in the number of tinyBees after some point does not contribute to the reduction in the execution time due to the cost of synchronization and number of kernel invocations.

7.9 Execution of Randomly Generated Tests

RQ5: How good is GVM performance when executing randomly generated tests?

Table 5 shows the results for running randomly generated tests. The list of classes used in each project are in the Appendix A. Each row shows results for one project: name of the project, number of tests, execution time with DJVM (DJVM), execution time with IJVM (IJVM), execution time with GVM when running one test in each GPU thread (GVM), and execution time with GVM when running one test in one warp (GVM-Warp). We show the last two columns to illustrate the impact of control flow divergence on the results and the benefit of the scheduling policy. In this set of experiments, we also expected that DJVM gains less from JIT, because each test is a different method executed only once. We note that the results in the Table include the startup cost, but we confirmed that this cost does not dominate the total execution time. Future work should study longer running tests and include more (and larger) projects, which will help to generalize our findings.

7.10 Thread Divergence and Memory Efficiency

RQ6: How do the thread divergence and memory efficiency impact GVM in various scenarios?

To understand how well GVM utilizes the parallel resources on the GPU, we characterize the control flow divergence, occupancy, and memory bandwidth utilization. Table 6 shows non-predicated warp execution efficiency (WNPEE), achieved occupancy (AO), and global load and

Table 6. GPU Efficiency Metrics for GVM's Engine Kernel, Averaged Across all Workloads for Shape Matching, Complete Matching, and Random Test Execution. The WNPEE Column is Non-predicated Warp Efficiency, AO is Achieved Occupancy, GLT and GST are Global Load and Store Throughput Respectively.

Workload	WNPEE	AO	GLT	GST
shape matching	70.5%	0.310	41 GB/s	42 GB/s
complete matching	59.8%	0.467	40 GB/s	145 GB/s
random test execution	9.0%	0.375	8760 MB/s	449 MB/s

store throughput (GLT, GST) as measured with `nvprof`. Data are averaged over all subjects for each experiment.

We take WNPEE as a proxy for control flow divergence: the closer WNPEE is to value 1, the *less* impact divergence has on performance. The data show that complete matching is 59.8% and shape matching is 70.5%. Average achieved occupancy for `engine` is 0.467 and 0.310 for complete and shape respectively, showing the SMs are approximately 1/3 utilized. Utilization is ultimately limited by memory capacity–because each TINYBEE requires significant global memory, this translates to creating fewer TINYBEEs than the execution resources of the hardware can support.

Both test generation techniques do essentially the same amount of work per thread in the `engine` kernel (each executing 10*e*6 instructions per warp). Shape matching and complete matching techniques use parallel sorting implemented by NVIDIA's `thrust::sort`, but do orders of magnitude less work in these kernels. So, while the other kernels have higher efficiency, utilization and memory throughput for shape and complete matching, the `engine` remains the dominant term for performance.

The WNPEE data for `randoop` are measured when GVM is running multiple TINYBEEs per warp, and the WNPEE of 9.0% shows that thread divergence is extreme, well below the 25% threshold for GVM to elect to run the workload with only a single hardware thread per warp. With a single thread per warp, WNPEE is still low because using only a single vector lane has low warp execution efficiency. However thread divergence disappears entirely, yielding higher overall performance.

7.11 Discussion of Dedicated Memory for CPU Runs

Finally, we briefly discuss the key reason to use 16GB of RAM for all IJVM and DJVM runs although our machine has 128GB of RAM. Our experiments on CPUs with 128GB (or any other value over 32GB) led to *worse* performance compared to runs with 16GB. This happens because Java uses 8 bytes instead of 4 for reference values, which hurts the performance [Vorontsov 2019]. Additionally, we decided to dedicate the same amount of memory to CPU runs and a single GPU; runs with more than 16GB and less than 32GB do not improve performance of IJVM or DJVM for our tasks.

8 LIMITATIONS AND FUTURE WORK

Java is huge: hundreds of pages for the language specification and runtime environment. GVM supports features necessary for many applications that compile to Java bytecode, but it omits many features that were not needed for the application targeted in this paper. We first report limitations that we encountered by running a benchmark for the Java language on top of GVM, then we document other limitations, and propose several directions for future work.

8.1 Limitations

To evaluate the state of GVM, we execute the test suites available in the K-Java [Bogdanas and Roşu 2015] on top of GVM [1]. K-Java targets the Java language rather than the JCL; the latter is

[1]We use K-Java available at github.com/kframework/java-semantics (SHA: `c202266`) with 834 programs in the test suite.

Table 7. Number of Programs in the K-Java Test Suites that GVM Passed and Failed.

Category	#Pass	#Fail	Category	#Pass	#Fail
smoke_tests	4	2	method_overloading	22	0
literals	9	1	method_access_mode	7	5
syntax_samples	5	0	fields	19	0
prim_operators	21	0	class_init	18	0
prim_conversions	14	0	constructors	19	0
string_plus	8	0	abstract	3	0
ref_operators	20	0	interfaces	24	0
op_cond_type	5	0	super_method	11	0
exp_type	29	0	static_methods	21	0
stmt_other	2	0	static_fields	11	0
stmt_loop	6	0	static_init	14	4
stmt_switch	8	0	interface_fields	25	8
stmt_return	20	0	static_init_trigger	24	7
stmt_throw	16	0	main_method	6	0
stmt_throw_thread_term	3	0	field_access_mode	4	1
stmt_break	21	0	comp_time_constants	5	0
stmt_continue	23	0	packages	34	0
arrays	40	8	class_literals	5	0
arrays_separated	35	7	inner_cl_static	48	1
basic_jvm_exc	2	2	inner_cl_instance	32	0
floats	7	13	local_cl	65	0
diverse	6	5	anonymous_cl	19	0
java_api_core	5	0	threads_run	0	7
method_basic	18	0	Σ	763	71

done in the Java Compatibility Kit (JCK). We used all programs available in the K-Java, and Table 7 shows the number of passing and failing programs organized by categories. GVM passes 763 tests and fails 71. However, all failed cases derive from scenarios GVM explicitly does not support:

a) 7 failures occur because we do not implement `monitorenter/monitorexit`.

b) 8 cases fail because they require reading command line inputs during execution, which GVM does not support.

c) 11 cases rely on very high-dimensional arrays (and GVM currently supports up to three dimensions) or handling rarely occurring array-related exceptions, e.g., `ArrayStoreException`.

d) 12 cases fail due to minor differences in floating point calculation between GPU and CPU.

e) 2 cases fail because GVM does not support initialization of interface fields via method calls.

f) 7 cases fail due to differences in string formatting.

g) 17 cases fail due to dynamic dispatch and static block initializers, e.g., GVM does not correctly discover an inherited package-private method if such a method is invoked from another package.

Of the 71 failing cases, 20 of them–b) and d)–are due to fundamental limitations of supporting a JVM on GPUs, while 51 simply require additional engineering effort. Other limitations, which we mentioned earlier in the paper, are due to our design decisions and focus on a specific application:

- GVM does not support multi-threaded programs, because our targeted application does not use multi-threading.

- GVM does not implement any actual GC; this decision was appropriate for testing workloads used in our evaluation, but could be a big limitation for other applications.
- GVM implements a subset of all native methods, i.e., those that were needed for used workloads.
- GVM does not support applications that require any access to external resources, such as the file system or network.

8.2 Future Work

Based on the results presented throughout this paper, we believe that GVM is an excellent base for several applications other than those used in this paper, including testing of software product lines [Kim et al. 2013; Nguyen et al. 2014] and implementing a symbolic execution engine [Anand et al. 2007; King 1976], which we plan to explore in the future.

We are also keen to extend our interpreter to support multi-threaded applications. This could be useful for several applications, including test generation for multi-threaded programs [Pradel and Gross 2012] and implementing a parallel software model checker [Godefroid 1997]. We plan to study two ways to model threads: (1) map each Java thread to a GPU thread (and use one TINYBEE per warp), or (2) enforce sequential execution with a single GPU thread with minimal number of context switches [Musuvathi and Qadeer 2007].

We focused on sequence-based test generation and test execution in this work, but other test generation algorithms, e.g., feedback-directed random test generation [Pacheco et al. 2007] could be parallelized by utilizing GVM; this will require minor modification to the system itself, but ensuring maximum sharing of control flow will be an interesting challenge.

GVM currently does not implement either runtime code optimizations nor ahead-of-time optimizations. Although these are current shortcomings of the system, they are also exciting opportunities that can substantially increase performance of GVM. In Section 7.6, we showed that ahead-of-time compilation can lead to performance improvement.

Our work in this paper was to understand how far we can get with a prototype Java bytecode interpreter that runs *entirely* on GPU. Future work should explore offloading only parts of application and utilizing both CPUs and GPUs *simultaneously*. This would be in line with prior work on offloading user annotated loops to a GPU [JCuda 2018; Pratt-Szeliga et al. 2012; Yan et al. 2009], but code running on a GPU in our case would also be Java bytecode that enables usage of JCL, exception handling, etc.

9 RELATED WORK

GVM is the first system that executes Java bytecode interpreters on GPUs and our goal was to move entire algorithms to GPU (i.e., no code is executed on CPU). Prior work has tried to offload only parts of (parallel) computation by *compiling* code to CUDA/OpenCL. Nevertheless, the design of GVM is inspired by prior research on offloading computation from high-level languages to GPUs. We also summarize prior work on using GPUs for speeding up testing.

High-level languages on GPUs: Rootbeer [Pratt-Szeliga et al. 2012] enables programming GPUs in Java. A developer implements the `Kernel` interface for code to be run on GPUs. The goal of Rootbeer is to improve programmability of GPUs. Similarly, JaBEE [Zaremba et al. 2012] requires that a developer implements the `GPUKernel` interface and focuses on speeding up *parts* of a Java application. Java-GPU [GPU 2019] compiles annotated loops (`@Parallel`) to CUDA and handles data transfers automatically.

PTask [Rossbach et al. 2011], StreamIt [Thies et al. 2002], Hydra [Weinsberg et al. 2008] and Pencil [Baghdadi et al. 2015] provide graph-based dataflow programming models for offloading

tasks to GPUs. Flextream [Hormati et al. 2009] is compilation framework for synchronous dataflow models that dynamically adapts applications to an FPGA, GPU, or CPU target architecture.

Many other systems provide GPU support in a high-level language: C++ [Gregory and Miller 2012], Java [Catanzaro et al. 2010; JCuda 2018; Pratt-Szeliga et al. 2012; Yan et al. 2009], MATLAB [Bispo et al. 2015; Prasad et al. 2011], Python [Catanzaro et al. 2010; Klöckner et al. 2012], C# [Farooqui et al. 2014; Rossbach et al. 2013], and Haskell [Chakravarty et al. 2011]. Some go beyond simple GPU API bindings, and provide support for compiling the high-level language to GPU code, none support a prototype Java interpreter on the GPU, and all expose the underlying device abstraction. Liszt's [DeVito et al. 2011] static meshes, Halide's [Ragan-Kelley et al. 2017] coordinate spaces, Lime's [Dubach et al. 2012] type system, Legion's [Bauer et al. 2012] logical regions, and Tian et al. [2017] LLVM extension provide runtimes and compilers with leverage for identifying automatically parallelizable code: all enable high level programs to be parallelized and compiled to different target architectures. Because GVM supports a lightweight Java bytecode interpreter per GPU thread, such compiler hints are unnecessary.

Sumatra [Oracle 2019d] was a project by the OpenJDK community to take advantages of GPUs and other accelerators; the original focus was on speeding up parallel code in a Java application via an extension of the stream API. Unlike Sumatra, which parallelizes only a very specific construct in an application, the goal of our work is to run an entire application/algorithm on GPU by developing a Java bytecode interpreter that runs on GPUs; so far, our main motivation was to speed up testing tasks, but we believe that other interesting tasks can benefit from our work, e.g., implementing a symbolic execution engine.

Maas et al. [2012] and Akram et al. [2016] offloaded garbage collection (GC) from Jikes RVM [Jikes RVM 2019] to GPUs and heterogeneous multicore architectures. Unlike their work, GVM supports running many Java bytecode interpreters on a GPU. While GVM currently does not include GC, our computation of the checksum of a program state can be extended to implement other/actual GC algorithms.

GVM runs independent Java bytecode interpreters per vector lane on the GPU: isolation across TINYBEES is a design goal for which we expect help from emerging support for fine-grain memory protection on GPUs [Ausavarungnirun et al. 2017, 2018; Power et al. 2014; Vesely et al. 2016].

Test generation and execution on GPUs: Celik et al. [2017] presented intKorat, a technique for bounded-exhaustive test input generation on GPUs. Their approach was to implement an existing algorithm in CUDA. Unlike intKorat, we developed a system of Java bytecode interpreters and used the system for sequence-based test generation and execution of randomly generated tests. Yaneva et al. [2017] utilized GPUs to accelerate testing for embedded software written in C/C++.

GPUexplore [Wijs and Bošnački 2014; Wijs et al. 2016] is a technique for performing state space exploration and model checking using GPUs.

GVM vs. prior work: GVM makes headway in a problem area where previous attempts have been less successful for three reasons. First, GVM's offload target is the entire application, rather than just parallel phases. This has an important implication: GVM has limited need to identify performance profitable data parallel phases and balance the gains of parallelism against the overheads of offload. In particular, data movement overheads can have lower impact on performance. In most data-parallel offloads, a large volume of data must be transferred to/from the GPU, potentially synchronously, potentially many times. While program inputs/outputs can still require transfer in GVM, the offload does not require multiple phases of high-volume duplex communication because GVM offloads the entire program. Second, while GVM implements a complete bytecode interpreter on the GPU, the design anticipates use in domains that are, in fact, amenable to GPU execution. This means that we anticipate workloads with substantial shared control flow across different runs of the same or

Table 8. Classes Under Test Used in Experiments with Randomly Generated Tests.

ds : Data structures used in our sequence-based test generation
BinTree
BinomialHeap
FibHeap
IntAVLTreeMap
IntRedBlackTree
TreeMap
jcl : https://docs.oracle.com/javase/8/docs/api/
java.util.ArrayList
java.util.HashMap
java.util.IntSummaryStatistics
java.util.LinkedList
java.util.Spliterators
java.util.StringTokenizer
vectorz : https://github.com/mikera/vectorz
mikera.indexz.GrowableIndex
mikera.indexz.Index
mikera.vectorz.BitVector
mikera.vectorz.Scalar
mikera.vectorz.Vector
mikera.vectorz.Vector1
mikera.vectorz.Vector2
mikera.vectorz.Vector3
apache : https://github.com/apache/commons-collections
org.apache.commons.collections4.list.FixedSizeList
org.apache.commons.collections4.list.TreeList
org.apache.commons.collections4.map.DefaultedMap
org.apache.commons.collections4.map.FixedSizeMap
org.apache.commons.collections4.map.LRUMap
org.apache.commons.collections4.map.LinkedMap
org.apache.commons.collections4.queue.CircularFifoQueue
org.apache.commons.collections4.set.MapBackedSet

similar application invocations, rather than across different threads in the same application. Our claim is that this is sufficiently general to serve the needs of an important class of applications, but *not* that it will generalize to all applications. Third, GPU architectures improved significantly since most of the previous works in this area. Importantly, GPU core clock speeds have approximately doubled in the last several years, which means that the performance penalty for control flow divergence is relatively lower. It is still a dominant term for performance, as our evaluation shows, however, sequential execution on a GPU is now much faster, which gives GVM headroom to tolerate levels of divergence that would have made offload unprofitable on previous architectures.

10 CONCLUSION

We described the design and implementation of GVM, a system that enables simultaneous execution of many Java bytecode interpreters on GPUs. Our work is the first to implement an interpreter for a popular general-purpose language (Java) in CUDA. GVM is ideal for applications that execute a large number of independent tasks, which share the same codebase and execute in similar amount of

time. GVM uses novel algorithms, scheduling, and data layout techniques to adapt to the massively parallel programming and execution model of GPUs. We applied GVM to sequence-based test generation and execution of randomly generated tests. Our evaluation shows that GVM can be more effective than existing Java interpreters, and as efficient as running parallel algorithms using several CPU threads with Oracle JVM with JIT, which has been engineered and optimized for over 20 years. We believe that GVM provides a platform that can be exploited for many other domain-specific tasks, used side-by-side with CPUs to increase throughput, and combined with JVMs running on CPUs to solve problems that could benefit from extra computational power.

A SUBJECTS OF RANDOMLY GENERATED TESTS

This section shows the details of subjects used in Section 7.9. Table 8 shows the list of classes under test for our experiments with randomly generated tests.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments. We thank Kush Jain, Joshua Landgraf, Darko Marinov, Karl Palmskog, Marinela Parovic, Zhiqiang Zang, and Chenguang Zhu for their feedback on this work. This work was partially supported by the US National Science Foundation under Grant Nos. CCF-1652517, CCF-1704790, and CNS-1846169.

REFERENCES

Shoaib Akram, Jennifer B Sartor, Kenzo Van Craeynest, Wim Heirman, and Lieven Eeckhout. 2016. Boosting the Priority of Garbage: Scheduling Collection on Heterogeneous Multicore Processors. *Transactions on Architecture and Code Optimization* 13, 1 (2016), 4.

Amazon. 2018. Amazon EC2 Elastic GPUs. https://aws.amazon.com/ec2/elastic-gpus/.

Saswat Anand, Corina S. Păsăreanu, and Willem Visser. 2007. JPF-SE: A Symbolic Execution Extension to Java Pathfinder. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 134–138.

Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and Onur Mutlu. 2017. Mosaic: A GPU Memory Manager with Application-transparent Support for Multiple Page Sizes. In *International Symposium on Microarchitecture*. 136–150.

Rachata Ausavarungnirun, Vance Miller, Joshua Landgraf, Saugata Ghose, Jayneel Gandhi, Adwait Jog, Christopher J. Rossbach, and Onur Mutlu. 2018. MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 503–518.

Azure. 2018. Azure Windows VM sizes - GPU. https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes-gpu.

Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F Donaldson, Jeroen Ketema, Javed Absar, Sven van Haastregt, Alexey Kravets, Anton Lokhmotov, Robert David, and Elnar Hajiyev. 2015. Pencil: A Platform-Neutral Compute Intermediate Language for Accelerator Programming. In *International Conference on Parallel Architecture and Compilation*. 138–149.

Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing Locality and Independence with Logical Regions. In *International Conference on High Performance Computing, Networking, Storage and Analysis*. 66:1–66:11.

João Bispo, Luís Reis, and João M. P. Cardoso. 2015. C and OpenCL Generation from MATLAB. In *Symposium on Applied Computing*. 1315–1320.

David Blythe. 2006. The Direct3D 10 system. *ACM Trans. Graph.* 25, 3 (2006), 724–734.

Denis Bogdanas and Grigore Roşu. 2015. K-Java: A Complete Semantics of Java. In *Symposium on Principles of Programming Languages*. 445–456.

K.J. Brown, A.K. Sujeeth, H.J. Lee, T. Rompf, H. Chafi, and K. Olukotun. 2011. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *International Conference on Parallel Architectures and Compilation Techniques*. 89–100.

Bryan Catanzaro, Michael Garland, and Kurt Keutzer. 2010. *Copperhead: Compiling an Embedded Data Parallel Language*. Technical Report UCB/EECS-2010-124. EECS Department, University of California, Berkeley. http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-124.html

Ahmet Celik, Sreepathi Pai, Sarfraz Khurshid, and Milos Gligoric. 2017. Bounded Exhaustive Test-Input Generation on GPUs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 94:1–94:25.

Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. 2011. Accelerating Haskell Array Codes with Multicore GPUs. In *Workshop on Declarative Aspects of Multicore Programming*. 3–14.

Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. 2011. Liszt: A Domain Specific Language for Building Portable Mesh-based PDE Solvers. In *International Conference for High Performance Computing, Networking, Storage and Analysis*. 9:1–9:12.

Christophe Dubach, Perry Cheng, Rodric Rabbah, David F. Bacon, and Stephen J. Fink. 2012. Compiling a High-level Language for GPUs: (via Language Support for Architectures and Compilers). In *Conference on Programming Language Design and Implementation*. 1–12.

Chucky Ellison and Grigore Roşu. 2012. An Executable Formal Semantics of C with Applications. In *Symposium on Principles of Programming Languages*. 533–544.

Naila Farooqui, Christopher J. Rossbach, Yuan Yu, and Karsten Schwan. 2014. Leo: A Profile-Driven Dynamic Optimization Framework for GPU Applications. In *International Conference on Timely Results in Operating Systems*. 5–5.

Patrice Godefroid. 1997. Model Checking for Programming Languages Using VeriSoft. In *Symposium on Principles of Programming Languages*. 174–186.

Google. 2018. Graphics Processing Unit (GPU): Leverage GPUs on Google Cloud for Machine Learning and Scientific Computing. https://cloud.google.com/gpu/.

Java GPU. 2019. *Java GPU Code Archive*. https://code.google.com/archive/p/java-gpu.

Kate Gregory and Ade Miller. 2012. *C++ AMP: Accelerated Massive Parallelism with Microsoft Visual C++*. Microsoft Press.

Akihiro Hayashi, Max Grossman, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2013. Accelerating Habanero-Java Programs with OpenCL Generation. In *International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. 124–134.

Amir Hormati, Yoonseo Choi, Manjunath Kudlur, Rodric M. Rabbah, Trevor N. Mudge, and Scott A. Mahlke. 2009. Flextream: Adaptive Compilation of Streaming Applications for Heterogeneous Architectures. In *International Conference on Parallel Architectures and Compilation Techniques*. 214–223.

Java Pathfinder. 2019. *Java Pathfinder Home Page*. https://github.com/javapathfinder/jpf-core.

JCuda. 2018. Java Bindings for CUDA. https://www.jcuda.org/jcuda/JCuda.html.

Jikes RVM. 2019. *Jikes RVM Home Page*. https://www.jikesrvm.org.

Andrew Kerr, Gregory F. Diamos, and Sudhakar Yalamanchili. 2009. A Characterization and Analysis of PTX Kernels. In *International Symposium on Workload Characterization*. 3–12.

Khronos. 2019. *OpenCL Overview*. https://www.khronos.org/opencl.

Chang Hwan Peter Kim, Darko Marinov, Sarfraz Khurshid, Don Batory, Sabrina Souto, Paulo Barros, and Marcelo d'Amorim. 2013. SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems. 257–267.

James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394.

Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. 2012. PyCUDA and PyOpenCL: A Scripting-based Approach to GPU Run-time Code Generation. *Parallel Comput.* 38, 3 (2012), 157–174.

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 Architecuture*. 451–460.

Martin Maas, Philip Reames, Jeffrey Morlan, Krste Asanović, Anthony D. Joseph, and John Kubiatowicz. 2012. GPUs as an Opportunity for Offloading Garbage Collection. In *International Symposium on Memory Management*. 25–36.

Madanlal Musuvathi and Shaz Qadeer. 2007. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *Conference on Programming Language Design and Implementation*. 446–455.

Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. 2014. Exploring Variability-aware Execution for Testing Plugin-based Web Applications. In *International Conference on Software Engineering*. 907–918.

NVIDIA. 2019. *CUDA Zone*. https://developer.nvidia.com/cuda-zone.

Oracle. 2019a. *Java SE at a Glance*. https://www.oracle.com/technetwork/java/javase/overview/index.html.

Oracle. 2019b. *JEP 318: Epsilon: A No-Op Garbage Collector*. https://openjdk.java.net/jeps/318.

Oracle. 2019c. *JNI APIs and Developer Guides*. https://docs.oracle.com/javase/8/docs/technotes/guides/jni.

Oracle. 2019d. *OpenJDK Project Sumatra*. http://openjdk.java.net/projects/sumatra.

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.

Shoumik Palkar, James J. Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimarjan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman P. Amarasinghe, Samuel Madden, and Matei Zaharia. 2018. Evaluating End-to-End

Optimization for Data Analytics Applications in Weld. *Proceedings of the VLDB Endowment* 11, 9 (2018), 1002–1015.

Jonathan Power, Mark D Hill, and David A Wood. 2014. Supporting x86-64 Address Translation for 100s of GPU Lanes. In *International Symposium on High Performance Computer Architecture*. 568–578.

Michael Pradel and Thomas R. Gross. 2012. Fully Automatic and Precise Detection of Thread Safety Violations. In *Conference on Programming Language Design and Implementation*. 521–530.

Ashwin Prasad, Jayvant Anantpur, and R. Govindarajan. 2011. Automatic Compilation of MATLAB Programs for Synergistic Execution on Heterogeneous Processors. In *Conference on Programming Language Design and Implementation*. 152–163.

Philip C. Pratt-Szeliga, James W. Fawcett, and Roy D. Welch. 2012. Rootbeer: Seamlessly Using GPUs from Java. In *International Conference on High Performance Computing and Communication*. 375–380.

Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2017. Halide: Decoupling Algorithms from Schedules for High-performance Image Processing. *Commun. ACM* 61, 1 (2017), 106–115.

Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. 2011. PTask: Operating System Abstractions to Manage GPUs As Compute Devices. In *Symposium on Operating Systems Principles*. 233–248.

Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. 2013. Dandelion: a Compiler and Runtime for Heterogeneous Systems. In *Symposium on Operating Systems Principles*. 49–68.

Rohan Sharma, Milos Gligoric, Andrea Arcuri, Gordon Fraser, and Darko Marinov. 2011. Testing Container Classes: Random or Systematic? 262–277.

William Thies, Michal Karczmarek, and Saman P. Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *International Conference on Compiler Construction*. 179–196.

Xinmin Tian, Hideki Saito, Ernesto Su, Jin Lin, Satish Guggilla, Diego Caballero, Matt Masten, Andrew Savonichev, Michael Rice, Elena Demikhovsky, Ayal Zaks, Gil Rapaport, Abhinav Gaba, Vasileios Porpodas, and Eric Garcia. 2017. LLVM Compiler Implementation for Explicit Parallelization and SIMD Vectorization. In *Workshop on the LLVM Compiler Infrastructure in HPC*. 4.

Jan Vesely, Arkaprava Basu, Mark Oskin, Gabriel H. Loh, and Abhishek Bhattacharjee. 2016. Observations and Opportunities in Architecting Shared Virtual Memory for Heterogeneous Systems. In *International Symposium on Performance Analysis of Systems and Software*. 161–171.

Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. 2003. Model Checking Programs. *Automated Software Engineering* 10, 2 (2003), 203–232.

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.

Mikhail Vorontsov. 2019. Java Performance Tuning Guide. http://java-performance.info/over-32g-heap-java.

Yaron Weinsberg, Danny Dolev, Tal Anker, Muli Ben-Yehuda, and Pete Wyckoff. 2008. Tapping into the Fountain of CPUs: On Operating System Support for Programmable Devices. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 179–188.

Anton Wijs and Dragan Bošnački. 2014. GPUexplore: Many-core On-the-fly State Space Exploration Using GPUs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 233–247.

Anton Wijs, Thomas Neele, and Dragan Bošnački. 2016. GPUexplore 2.0: Unleashing GPU Explicit-state Model Checking. In *International Symposium on Formal Methods*. 694–701.

Yonghong Yan, Max Grossman, and Vivek Sarkar. 2009. JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA. In *European Conference on Parallel Processing*. 887–899.

Vanya Yaneva, Ajitha Rajan, and Christophe Dubach. 2017. Compiler-assisted Test Acceleration on GPUs for Embedded Software. In *International Symposium on Software Testing and Analysis*. 35–45.

Wojciech Zaremba, Yuan Lin, and Vinod Grover. 2012. JaBEE: Framework for Object-oriented Java Bytecode Compilation and Execution on Graphics Processor Units. In *Workshop on General Purpose Processing with Graphics Processing Units*. 74–83.