Programming and Execution Models for Parallel Bounded Exhaustive Testing

NADER AL AWAR, The University of Texas at Austin, USA
KUSH JAIN, The University of Texas at Austin, USA
CHRISTOPHER J. ROSSBACH, The University of Texas at Austin and Katana Graph, USA
MILOS GLIGORIC, The University of Texas at Austin, USA

Bounded-exhaustive testing (BET), which exercises a program under test for all inputs up to some bounds, is an effective method for detecting software bugs. Systematic property-based testing is a BET approach where developers write test generation programs that describe properties of test inputs. Hybrid test generation programs offer the most expressive way to write desired properties by freely combining declarative filters and imperative generators. However, exploring hybrid test generation programs, to obtain test inputs, is both computationally demanding and challenging to parallelize. We present the first programming and execution models, dubbed Tempo, for parallel exploration of hybrid test generation programs. We describe two different strategies for mapping the computation to parallel hardware and implement them both for GPUs and CPUs. We evaluated Tempo by generating instances of various data structures commonly used for benchmarking in the BET domain. Additionally, we generated CUDA programs to stress test CUDA compilers, finding four bugs confirmed by the developers.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; • **Computing methodologies → Parallel programming languages**.

Additional Key Words and Phrases: Bounded exhaustive testing, Test generation, Parallel programming

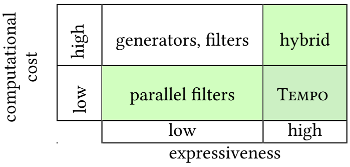
ACM Reference Format:
Nader Al Awar, Kush Jain, Christopher J. Rossbach, and Milos Gligoric. 2021. Programming and Execution Models for Parallel Bounded Exhaustive Testing. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 166 (October 2021), 28 pages. https://doi.org/10.1145/3485543

1 INTRODUCTION

Automated test generation has been shown effective for discovering software bugs [Daniel et al. 2007; Gligoric et al. 2010; Selakovic et al. 2018; Yang et al. 2011; Zeller et al. 2019; Zhang et al. 2017]. *Bounded exhaustive testing* (BET), which is based on a small-scope hypothesis [Jackson and Damon 1996], systematically checks the correctness of a program under test for all possible *test inputs* up to the user-specified bounds. In BET, a developer writes a *test generation program* and then *explores* that program to obtain test inputs.

Authors' addresses: Nader Al Awar, The University of Texas at Austin, Austin, TX, 78712, USA, nader.alawar@utexas.edu; Kush Jain, The University of Texas at Austin, Austin, TX, 78712, USA, kjain14@utexas.edu; Christopher J. Rossbach, The University of Texas at Austin and Katana Graph, Austin, TX, 78712, USA, rossbach@cs.utexas.edu; Milos Gligoric, The University of Texas at Austin, Austin, TX, 78712, USA, gligoric@utexas.edu.

2475-1421/2021/10-ART166
https://doi.org/10.1145/3485543



		low	high
computational cost	high	generators, filters	hybrid
	low	parallel filters	Tempo
		expressiveness	

In *systematic property-based testing* [Boyapati et al. 2002; Gligoric et al. 2010; Kuraj et al. 2015; Rosner et al. 2014], a widely studied BET approach, a developer writes test generation programs by specifying desired properties of test inputs. Prior work has proposed several ways to write these properties. First, a developer can specify properties of test inputs by writing *declarative filters* [Boyapati et al. 2002] (e.g., `isAcyclic(bst) && areKeysOrdered(bst)`) that encode properties that each test input has to satisfy. Second, a developer can write *imperative generators* [Daniel et al. 2007; Kuraj et al. 2015] (e.g., `bst = constructTree(); assignKeys(bst)`) that describe how to construct each test input of interest. Third, a developer can mix declarative filters and imperative generators, also known as the *hybrid* style [Gligoric et al. 2010; Rosner et al. 2014] (e.g., `bst = constructTree(); areKeysOrdered(bst)`). Among the three, the hybrid style provides the programmer with the most expressive power, but is computationally costly due to the large space that has to be explored to obtain the desired test inputs.

Several approaches were developed for parallel exploration of filtering [Celik et al. 2017; Misailovic et al. 2007] and generating [Kuraj et al. 2015] test generation programs. However, these approaches are algorithm-specific and provide no generic mechanism that would support hybrid test generation programs.

In this paper, we present a *programming model* and *execution model* for parallel exploration of hybrid test generation programs called Tempo. In the Tempo programming model, all tests are generated in parallel based on a simple API comprising three functions. The Tempo execution model determines how parallel generation work is mapped to physical hardware. A key primitive in Tempo, inspired by UDITA [Gligoric et al. 2010], is a *non-deterministic choice*. In contrast to UDITA, Tempo leverages non-deterministic choice to automatically parallelize the exploration. Furthermore, Tempo automatically balances the resource usage (threads and memory) to enable efficient exploration for large bounds.

We implemented a runtime for Tempo that has two novel execution *strategies*–re-execution based and fork based–each supported on two backends–GPUs (with CUDA) and CPUs (with OpenMP). In the *re-execution based strategy*, each non-deterministic choice halts the execution of a task and spawns a group of new tasks. Each new task will then re-execute the test generation program from the beginning. Once a task arrives at the non-deterministic choice that spawned that task, it will select a unique value and continue execution. In the *fork based strategy*, the total number of tasks is estimated from the test generation program and remains constant during exploration; each non-deterministic choice splits the tasks into multiple groups, which is equal to the number of choices.

We used Tempo to write test generation programs for dozens of data structures, commonly used for benchmarking in the BET domain [Boyapati et al. 2002; Celik et al. 2017; Kuraj et al. 2015; Misailovic et al. 2007; Rosner et al. 2014; Visser et al. 2006]. We show the pros and cons of each of our strategies and analyze their performance in detail both on GPUs and CPUs. Our results show that on GPUs, the fork based strategy performs better for test generation programs that follow simple patterns; otherwise the re-execution based strategy is preferred. We applied Tempo to the Nvidia CUDA compiler (NVCC) by writing several test generation programs to generate inputs for compilers; then, we fed the inputs to three compilers—NVCC, GCC, and Clang—and used differential testing [McKeeman 1998] to compare their results. So far, the generated inputs have revealed four bugs, with two bugs in NVCC being confirmed as new. The main contributions of this paper include:

Table 1. List of Functions Provided by Tempo's API.

Function	Description
`_explore(void (*k)(void*), void *args)`	Explores an entire test generation program that starts with kernel `k`
`_choice(int min, int max)`	Non-deterministically chooses a value between `min` and `max`, inclusive; precondition: `max` is greater or equal to `min`
`_ignoreIf(boolean condition)`	Stops the execution of the current thread if the given condition holds

- **Programming and execution models**. We present the first programming and execution models for parallel exploration of hybrid test generation programs. Thus, Tempo enables efficient exploration for the most expressive BET style. Tempo automatically manages memory and controls parallelism by managing threads.
- **Strategies and platforms**. We provide a runtime with two strategies for Tempo: re-execution based (that spawns tasks on demand) and fork based (that estimates the number of tasks prior to an exploration and splits the tasks as necessary during the exploration). We implemented both strategies for GPUs and CPUs, which enabled us to study their complementary strengths and weaknesses. We also show a way in which they can be combined.
- **Evaluation**. We evaluated Tempo using a large set of data structures. Our results show that the fork-based strategy is an excellent choice for test generation programs that follow simple structure and run on GPUs, while in other cases the re-execution based strategy performs better. Additionally, we perform a case study in which we designed several generators for testing the NVCC compiler.

Artifacts related to Tempo are publicly available at https://github.com/EngineeringSoftware/tempo.

2 PROGRAMMING MODEL

In this section, we explain Tempo's programming model and show how to use Tempo to write test generation programs. Although our focus in this paper is on hybrid test generation programs [Gligoric et al. 2010; Rosner et al. 2014], Tempo subsumes several other styles (e.g., [Daniel et al. 2007]). We also show an example written in the sequence-based style [Visser et al. 2006; Xie et al. 2005] to demonstrate the generality of Tempo.

2.1 API

Table 1 lists functions available to developers in the API. The first column shows function signatures and the second column provides a brief description of each function.

A call to `_explore(tgp, args)` starts the exploration of a test generation program (i.e., a kernel function) that is given as the first argument; the second argument (`args`) is passed to each kernel invocation by our runtime. `_explore` is the only function from our API that always executes on a CPU. The function finishes execution when all possible paths in the test generation program are fully executed. We will see a use of this function in Figure 1c (line 3).

A call to `_choice(min, max)` creates a non-deterministic *choice* that requires that the program is executed (from the invocation point) for each value between `min` and `max` (inclusive). A straightforward way to implement this is to execute programs sequentially from the beginning (i.e., re-execute) and choose a different value for one of the `_choice` invocations each time. Our execution strategies–fork based (ForkStrategy) and re-execution based (ReexeStrategy)–differ in the way they implement `_choice`.

```
1  void nodeUpdate(Node *node) {
2    node->value = _choice(0, MAX_VALUE); }
3
4  void initSubTree(Node *node, int size) {
5    int left_size = _choice(0, size);
6    int right_size = size - left_size;
7    if (left_size != 0) {
8      node->left = nodeAlloc();
9      nodeUpdate(node->left);
10     initSubTree(node->left, left_size - 1); }
11   if (right_size != 0) {
12     node->right = nodeAlloc();
13     nodeUpdate(node->right);
14     initSubTree(node->right, right_size - 1); }}

15 void generate(BT *bt, int size) {
16   bt->size = size;
17   bt->root = nodeAlloc();
18   nodeUpdate(bt->root);
19   initSubTree(bt->root, bt->size - 1);
20   bool is_ordered = orderProperty(bt);
21   _ignoreIf(!is_ordered); }
22
23 __global__ void tgp(void *arg) {
24   int size = (int) arg;
25   // ... MAX_VALUE = size - 1
26   BT bt;
27   generate(&bt, size); }
```

(a) Hybrid style

```
__global__ void tgp(void *arg) {
  int seq_len = (int) arg;
  BT bt;
  for (int i = 0; i < seq_len; i++) {
    int p = _choice(0, 1);
    int v = _choice(0, seq_len-1);
    switch (p) {
    case 0: bt.add(v); break;
    case 1: bt.remove(v); break; }}}
```

(b) Sequence-based style

```
int main(int argc, char *argv[]) {
  // N = parse from argv
  _explore(&tgp, (void*) N);  }
```

(c) Main program to start a test generation program

Fig. 1. Examples of test generation programs for a Binary Search Tree (BT) data type.

A call to `_ignoreIf(condition)` stops the execution if the given condition evaluates to `true`; otherwise this function call has no effect. This function can be used for an early stop of any path that has generated a test input that violates a property of the structure being generated while executing a test generation program.

2.2 Hybrid Test Generation Programs

Test generation programs that mix filters and imperative generators encode both the set of properties that each generated test input has to satisfy (with *filters*) and how to create a part of each instance of interest (with *generators*). Figure 1a shows an example of a test generation program that generates valid binary search trees up to the given size; the program looks very much the same as in the original work on hybrid test generation programs [Gligoric et al. 2010]. The key difference is that Tempo automatically parallelizes the exploration of this test generation program and manages resource usage for large sizes.

The `generate()` function (lines 15-21) calls `initSubTree()` (line 19) which recursively constructs both left and right subtrees of the binary tree being generated. The call to `_choice` in `initSubTree()` (line 5) selects the size of both left and right subtrees of the binary tree. We use `_choice` to *non-deterministically* choose a value. Tempo guarantees that *every possible combination of values* returned by all calls to `_choice` will be executed. `nodeUpdate()` assigns a value to a node by also calling `_choice`. Finally (line 20), `generate()` checks whether the instance generated by `initSubtree()` is a valid binary tree by calling the `orderProperty()` filter. The `orderProperty()` filter checks if a key in a node is larger than all the keys in the left subtree and smaller than the keys in the right subtree; we do not show this function because it does not contain features relevant for this paper. `_ignoreIf()` is used to conditionally terminate threads.

2.3 Sequence-Based Test Generation Programs

Sequence-based testing [Visser et al. 2006] systematically explores all combinations of function calls for the system under test up to a given bound. Figure 1b shows an example of a test generation program for the Binary Search Tree class (`BT`).

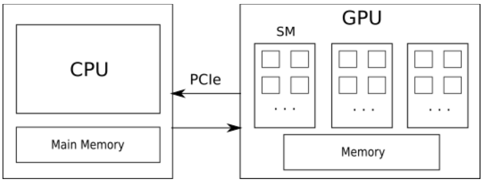
In each iteration of the loop (lines 4-9), the program non-deterministically chooses a function to invoke (line 5) and the argument for the function (line 6), and then invokes the selected function with the selected argument. Figure 1c shows the main function (executed on the CPU) that invokes `_explore` from Tempo's API, which triggers the exploration of the given test generation program for the maximum sequence length (`N`). For example, if we execute the program for `N = 2`, Tempo would explore the following sequences: (1) `t.add(0)`, `t.add(0)`; (2) `t.add(0)`, `t.add(1)`; (3) `t.add(0)`, `t.remove(0)`; (4) `t.add(0)`; `t.remove(1)`; ... (16) `t.remove(1)`, `t.remove(1)`.

3 EXECUTION MODEL

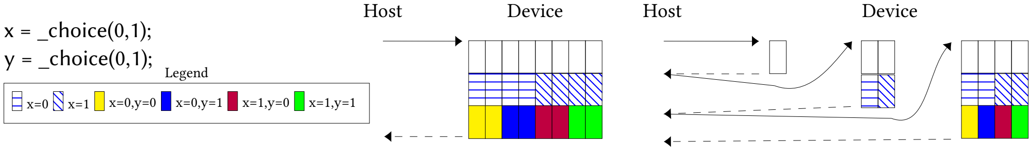
This section describes the details of our execution model. First, we provide a brief background on CUDA (Section 3.1) and introduce terminology used in later sections. We also articulate the motivation for using GPUs as one of the target platforms for exploration of hybrid test generation programs. Second, we provide a high-level overview of the framework (Section 3.2), then give semantics for functions in the Tempo API (Section 3.3). We describe two strategies that implement the execution model (Section 3.4) in a runtime, as well as an in-built mechanism for supporting exploration of test generation programs for large bounds (Section 3.5). We discuss strengths and weaknesses of the two algorithms (Section 3.6) and the way to combine them (Section 3.7).

3.1 Background on CUDA

The figure on the right illustrates a GPU-enabled system in which a discrete GPU is connected to a CPU via a PCIe link. An Nvidia GPU comprises a number of streaming multiprocessors (SMs) each with dozens of arithmetic pipelines. SMs are capable of concurrent execution of thousands of *GPU threads* (threads for short). Each discrete GPU has its own memory that is directly addressable by the SMs, and which is attached with high bandwidth interconnect enabling transfer rates of 100–1000s of GB per second [CUDADocs 2020].



A program written in CUDA [CUDAWebPage 2020] combines code that is executed on a CPU and a GPU. A function that can be executed on a GPU and invoked from the CPU has to be annotated with `__global__`; these functions are called *kernels*. We showed one example of a kernel in Figure 1b. Usually, a GPU application would (1) prepare data on the CPU, (2) transfer the data to the GPU memory, (3) execute one or *more* kernels, and (4) transfer the resulting output data back to the main memory for further processing. An invocation of a kernel (`kernel_name«<num_blk,num_th»>(arguments)`) looks like a normal function invocation, but it also configures the number of *blocks* (`num_blk`) and the number of threads (`num_th`) in each block; the total number of threads is `num_blk` × `num_th`. Each thread has a unique identifier (`tid`). Blocks are assigned to SMs that can run multiple blocks concurrently using hardware-supported multi-threading. Ideally, the spawned threads oversubscribe the SMs, which have an efficient mechanism for context switching to hide memory latency. On NVIDIA hardware, threads are grouped into warps that have 32 threads each. All threads within a warp *execute the same instruction at any given time*. Thus, any conditional



(a) Sample Program (b) ForkStrategy (c) ReexeStrategy

Fig. 2. High-level idea behind Tempo's algorithms. Each column corresponds to a thread and each row shows the way an invocation of `_choice` impacts program state for each thread. Cells with same colors and patterns indicate that threads are in the same program state.

branch (e.g., an `if` statement) in code can have a substantial negative impact on performance, as it requires executing both branches in sequence, by effectively suppressing the effects of threads that enter one branch while the other branch is being executed.

We find that CUDA and GPUs are a promising target for exploring test generation programs for (at least) the following several reasons. One, test generation programs can require execution of hundreds of thousands of *program paths* which can be handled in parallel, making massively parallel GPUs an appropriate platform. Two, an exploration of a single test generation program executes many paths that share common prefixes, which reduces branch divergence and increases the likelihood that all threads in a warp execute the same control flow. Three, test generation programs based on property-based testing are primarily used for generating test inputs. Although test generation programs may be complex, they do not require that the program under test be executed in the same environment. Specifically, we can use GPUs for test input generation, but we do not need to use them for running the program under test. One potential concern is that test execution time will dwarf generation time, thereby minimizing the impact of Tempo. However, test execution itself can be optimized (e.g., using parallelism) to the point where generation is the bottleneck. We illustrate this in Section 6. However, a full exploration of speeding up test execution is beyond the scope of this paper.

In this work we implemented our strategies on both GPUs and CPUs. Although we use both platforms, our goal is not to establish which is better, only to show that our technique can be implemented on both.

3.2 Overview of the Execution Model

Figure 2 shows an overview of Tempo's strategies: *ForkStrategy* and *ReexeStrategy*; the subfigures illustrate the strategies for an example program (Figure 2a). The input to Tempo, regardless of the chosen strategy, is a test generation program similar in style to the examples described in Section 2. We use *device* to refer to the processor that runs the kernel in parallel, i.e. a GPU using CUDA or a CPU using OpenMP, and *host* to refer to the processor that calls the kernel. Both strategies execute partially on the host and partially on the device.

Once started, ForkStrategy (Figure 2b) spawns a large number of threads that start executing the same code. When threads encounter `_choice`, they split into groups (the number of groups depends on arguments given to `_choice`); threads in the same group have the same (albeit independent) program state. In ReexeStrategy (Figure 2c), the exploration starts with a single thread. Once the thread encounters `_choice` it terminates and new threads are scheduled. The number of new threads is equal to the number of choices in the latest `_choice` invocations.

3.3 Semantics

In figures 3 and 4, we show the semantics for the functions defined in the API for ForkStrategy and ReexeStrategy, respectively. (We assume standard semantics for common language constructs, e.g.,

```
1. _explore(K, A) ⟶ for (i in 0..#items) spaw(K, A, ⟨K, A, i, #items⟩)
2. _choice(min, max) ⟨ws, ···⟩ ⟶ if (ws < #choices) abort() else split(min, max) ⟨ws, ···⟩
3. split(min, max) ⟨ix, ws, ···⟩ ⟶ val = min + ⌊ix/⌊ws/#choices⌋⌋;
   if (val > max) exit(0) else val ⟨ix%⌊ws/#choices⌋, ⌊ws/#choices⌋⟩
4. _ignoreIf(condition) ⟨···⟩ ⟶ if (condition) exit(0) ⟨···⟩
```

Fig. 3. Rewrite rules for the API functions from Table 1 for ForkStrategy. #choices is equal to `max-min+1`.

```
1. _explore(K, A) ⟶ spaw(K, A, ⟨K, A, 0, 0, {}⟩)
2. _choice(min, max) ⟨t, ···⟩ ⟶ if (ix < t) _spawrange(min, max) else _advance() ⟨t, ···⟩
3. _advance() ⟨ ix, ···⟩ ⟶ M[ix] ⟨ix+1, ···⟩
4. _spawrange(min, max) ⟨···⟩ ⟶ for (i in min..max) _spawone(i); exit(0) ⟨···⟩
5. _spawone(i) ⟨ K, A, t, ix, M⟩ ⟶ spaw(K, A, ⟨K, A, t+1, 0, M[ix/i]⟩) ⟨···⟩
6. _ignoreIf(condition) ⟨···⟩ ⟶ if (condition) exit(0) ⟨···⟩
```

Fig. 4. Rewrite rules for the API functions from Table 1 for ReexeStrategy.

`if` statements.) We use rewrite rules in a style similar to the one used by Maude and K rewriting engines [Ellison and Roşu 2012; Meseguer and Roşu 2013]. In each rule, we show only the relevant components of the configuration and we use ··· to denote the unchanged parts; components are always listed in the same order on the left and right sides for each rule.

Our rewrite rules for both strategies (ForkStrategy and ReexeStrategy) use a function primitive `spaw`. This primitive schedules a new *task* for execution. In this section, we refer to a task as a unit of work that needs to be executed, and we show how we map tasks to (CUDA and OpenMP) threads in Section 3.4. Our rewrite rules are generic and can be implemented in (m)any parallel frameworks, e.g., CUDA [CUDAWebPage 2020], OpenMP [OpenMPWebPage 2020], Legion [Bauer et al. 2012], and MPI [OpenMPIWebPage 2020]. We also note that the order in which tasks are executed does not matter for successful exploration, although the way in which they are *scheduled for execution* is critical to achieving good performance, e.g., when running on a GPU.

3.3.1 ForkStrategy. We define the configuration for ForkStrategy as ⟨`K`,`A`,`ix`,`ws`⟩, where `K` is the test generation program being explored, `A` are the arguments given to the test generation program, `ix` is the index of a task within a *task group*, and `ws` is the size of the task group.

Rule 1 (Figure 3) spawns `#items` tasks, such that each task has a unique index and they all belong to one task group (whose size is `#items`). Initially, all tasks start executing the kernel from the beginning and follow the same program path. Rule 2 shows how an invocation of `_choice` impacts the configuration of each task. If the number of tasks in the group executing `_choice` is not sufficient to cover all values in the choice (i.e. `max-min+1`), the execution terminates unsuccessfully; otherwise, we use a function `split` (Rule 3) to divide all the tasks within a task group into `max-min+1` different new task groups. Each task group is assigned a unique value in the range `[min,max]` inclusive, i.e., tasks in the same task group will continue executing the same program path. Additionally, `split` updates the `ix` of each task to reflect its index within the new group, such that the first task in the group always has `ix==0`. Rule 4 is self-explanatory.

3.3.2 ReexeStrategy. We define the configuration for ReexeStrategy as ⟨`K`,`A`,`t`,`ix`,`M`⟩, where `K` is the test generation program being explored, `A` are the arguments given to the program, `t` is the total number of known choices, `ix` is the index of the next choice, and `M` is memory that maintains a map from indices to values. We define the memory update `M[ix/val]` to change the value associated with the index `ix` to `val` without changing any other memory location.

Rule 1 (Figure 4) runs a single task and initializes all entries to zero values (i.e., no `_choice` invocations have been seen). Rule 2 shows what happens at each `_choice` invocation. When a `_choice` is executed for the first time (Rule 4), we create `max-min+1` new tasks, e.g., when the

```
Require: tgp - test generation program
Require: args - user provided arguments for the test generation program
 1: function EXPLOREFORK(tgp, args)
 2:     ethreads ← global.should_estimate ?
 3:         ESTIMATE(tgp, args, global.num_threads_during_estimate) :global.LARGE_NUMBER
 4:     INITMETADATAONGPU(ethreads)
 5:     iteration_count ← 0
 6:     while ethreads > 0 do
 7:         iteration_count ← iteration_count + 1
 8:         global.offset ← global.chunk_size ∗ iteration_count
 9:         currentThreads ← MIN (ethreads, global.chunk_size )
10:         threads ← GETNUMTHREADS(currentThreads)
11:         blocks ← GETNUMBLOCKS(currentThreads)
12:         tgp⟨⟨⟨blocks, threads⟩⟩⟩(args)
13:         ethreads ← ethreads − currentThreads
14:     end while
15: end function
```

Fig. 5. Algorithm behind the `_explore` function as implemented in ForkStrategy. This function is executed on the host. The highlighted code is needed for scaling the algorithm.

original task encounters the first choice. Each newly created task (Rule 5) inherits everything from the current task and additionally gets assigned one unique value in `[min,max]` inclusive for the current `_choice` invocation. Note that new tasks start executing the test generation program *from the beginning* and the current task terminates. In the second case (Rule 3), the current task already knows the value for the `_choice` invocation, so it reads the value from the memory (Rule 3) and continues. Similar to ForkStrategy, Rule 6 is self-explanatory.

3.4 Algorithms for Execution Strategies

Next, we describe algorithmic details of our execution strategies that follow semantics from the previous section. As mentioned before, each strategy executes partially on the host and partially on the device. Host execution involves memory allocation and initialization and configuring and launching kernels. Device execution includes parallel execution of kernels and assigning return values of `_choice` to different threads. We characterize the algorithms for GPUs, as the implementation for CPUs is much simpler.

3.4.1 ForkStrategy. Our key insight is to enable forking by spawning a very large number of threads initially (that execute in a lockstep) and then diverging their execution when forking is needed. When a new `_choice` invocation is encountered, ForkStrategy splits the threads that called `_choice` into a number of new task groups, with each new task group being assigned a unique value (in the [`min`,`max`] range) that is returned to all threads in that task group. In this algorithm, each task is mapped to a thread, and the number of tasks is equal to the number of threads.

Host. The `ExploreFork` function in Figure 5 is always executed on the host; highlighted lines are explained in Section 3.5.1. The algorithm accepts a test generation program and user-provided arguments to that program. Line 4 allocates and initializes memory on the device to hold each thread's metadata (`tid` and group size); each thread has a unique index and they all belong to the same group. The rest of the function calculates how many blocks and threads are needed (lines 10 and 11) and launches the kernel (line 12).

Device. The `_choiceFork`function in Figure 6 is executed on the device. First, a thread gets the group size of the task group it belongs to (line 4) and its index in that group (line 5). These two

```
Require: $min$ - the min value to choose
Require: $max$ - the max value to choose
 1: function _choiceFork($min, max$)
 2:     $tid \leftarrow$ getTid()
 3:     $tid \leftarrow tid + global.\text{offset}$
 4:     $groupSize \leftarrow$ getGroupSize($tid$)
 5:     $threadIndex \leftarrow$ getThreadIndex($tid$)
 6:     $numChoices \leftarrow max - min + 1$
 7:     if $groupSize < numChoices$ then
 8:         exit
 9:     end if
10:     $newGroupSize \leftarrow groupSize/numChoices$
11:     $newThreadIndex \leftarrow threadIndex\%newGroupSize$
12:     setGroupSize($tid, newGroupSize$)
13:     setThreadIndex($tid, newThreadIndex$)
14:     $choiceValue \leftarrow min + (threadIndex/newGroupSize)$
15:     if $choiceValue > max$ then
16:         exit
17:     end if
18:     return $choiceValue$
19: end function
```

Fig. 6. Algorithm behind the _choice function as implemented in ForkStrategy. This function is executed on the device. The highlighted code is needed for scaling the algorithm.

```
Require: $min$ - the min value to choose
Require: $max$ - the max value to choose
function _choiceEstimate($min, max$)
    $tid \leftarrow$ getTid()
    $numChoices \leftarrow max - min + 1$
    setEstimate($tid$, getEstimate($tid$) $* numChoices$)
    if $tid == 0$ then
        return $min$
    else if $tid == totalThreads - 1$ then
        return $max$
    end if
    return $min +$ rand()$\%(max - min + 1)$
end function
```

Fig. 7. Algorithm behind the _choice function during estimation as implemented in ForkStrategy.

values are obtained from the metadata initialized in ExploreFork. Then, it calculates the number of integer values in [min,max] (line 6). This value determines how many new task groups are needed. In order for each new task group to have at least one thread, there needs to be *at least* numChoices different threads in the current task group; otherwise, execution halts (line 8). If there are enough threads available, the thread calculates the size of the new task groups and its index in its new task group and updates those values (lines 10-13). It then determines the return value by dividing its index in the initial task group with the number of threads per each new task group, and adding min to that (line 14). This maps the value of each new task group to a unique value in [min,max].

Some threads might not be placed in a new task group due to truncation resulting from the integer division. These threads are simply terminated (line 16).

Estimating number of threads. A key aspect is how many threads to start with (ethreads, short for estimated threads); we used a global parameter (line 3 in Figure 5) to decide if we run a fixed number of threads (LARGE_NUMBER) or estimate the number of threads needed. Running a larger number of threads than necessary has a negative performance impact on ForkStrategy. The overhead introduced by these additional, *redundant*, threads is significant as they will still execute the test generation program without contributing to the exploration. The impact of this overhead is evident when generating structures of a relatively small size. Selecting the right value for the total number of threads is therefore important. A small constant could result in better performance, but will not work when exploring a relatively large number of paths. A large constant works for many paths, but redundant threads will negatively impact performance.

Since the actual number of paths can only be determined after the test generation program has been fully explored, we try to *estimate* the number of paths by running a *partial exploration* first. We execute (Figure 5, line 3) the test generation program by launching a single kernel with a relatively small number of threads (set to 10k in our experiments, although this value is configurable).

Figure 7 shows how we implement _choice during partial exploration. First, a thread obtains its tid and calculates the number of choices that can be returned (lines 2 and 3). Then it updates its estimate by multiplying it with numChoices (line 4); initially all estimates are set to 1 (not shown in the algorithm). Finally, a value needs to be returned. Lines 5-10 show how that value is calculated. If the calling thread's tid is 0, the minimum value is returned. If its tid is equal to num_threads_during_estimate-1 (the maximum possible tid), the maximum value is returned. Otherwise, a value between min and max is selected. This ensures that we explore both the minimum and maximum *bounds*.

3.4.2 ReexeStrategy. This strategy is based on the idea of *re-executing the kernel* as many times as necessary in order to fully explore a test generation program (recall Figure 2c). This strategy uses the exact number of threads needed to cover all values for all encountered _choice invocations.

The configuration in the semantics rules (Section 3.3) describes a task on an abstract level, and here we focus on the actual representation. A task describes one path in the exploration of a test generation program; thus, each task is assigned to a single thread and has a unique combination of values returned by _choice. A task is represented in memory as a fixed-length array. The total number of _choice invocations (on the current path) is stored at index 0, and the index of the current _choice is stored at index 1. The rest of the array stores the values of calls to _choice.

Host. Figure 8 shows the part of ReexeStrategy executed on the host; we describe highlighted lines in Section 3.5.2. This code accepts as input a test generation program and user-provided arguments to be passed to each program invocation. The algorithm first initializes the worklists (lines 2-3), and then creates a single task and includes it into the input worklist (line 5). The input worklist, which is in device memory, is an array that stores tasks; we use an array representation to optimize memory accesses. The initial task is "empty", i.e., the number of _choice invocations on the path to be executed is 0 (and the execution of this task will end upon the first call to _choice).

Next, the algorithm enters a loop (lines 6-14) that terminates when no more tasks are available in the input worklist. In each iteration, the algorithm determines the number of threads and blocks based on the number of tasks in the input worklist (lines 7-9), invokes the kernel with the determined number of threads (line 10), and waits for all the threads to finish their execution. If a thread encounters an invocation of _choice, it creates new tasks in the output worklist and ends its execution; the details are shown later when we talk about the part of the algorithm that runs on the device. Once all the threads finish their execution, either because they reached the end of the path or because a new _choice was encountered, the algorithm moves the tasks from the output worklist to the input worklist (line 11) and goes into the next iteration of the loop.

Device. Figure 9 provides the algorithmic view of the _choice function. Initially (lines 2-3) a thread invoking _choice obtains pointers to the input worklist and the output worklist; these pointers are

```
Require: tgp - test generation program
Require: args - user provided arguments for the test generation program
function exploreReexe(tgp, args)
    $inWL \leftarrow \emptyset$
    $outWL \leftarrow \emptyset$
    $overflowBuf \leftarrow \emptyset$
    append(inWL, (0, 0, []))
    do
        $inWLSize \leftarrow$ getWLSize(inWL)
        $threads \leftarrow$ getNumThreads(inWLSize)
        $blocks \leftarrow$ getNumBlocks(inWLSize)
        $tgp\langle\langle\langle \text{blocks}, threads \rangle\rangle\rangle(args)$
        moveFromTo(outWL, inWL)
        moveFromTo(outWL, overflowBuf)
        moveFromTo(overflowBuf, inWL)
    while getWLSize(inWL) $> 0$
end function
```

Fig. 8. Algorithm behind the _explore function as implemented in ReexeStrategy. This function is executed on the host. The highlighted code is needed for scaling the algorithm.

```
Require: min - the min value to choose
Require: max - the max value to choose
 1: function _choiceReexe(min, max)
 2:     $inWL \leftarrow$ getInputWorklist()
 3:     $outWL \leftarrow$ getOutputWorklist()
 4:     $tid \leftarrow$ getTid()
 5:     $total \leftarrow$ getNumOfKnownChoices(inWL, tid)
 6:     $index \leftarrow$ getCurrentChoiceIndex(inWL, tid)
 7:     if $index >= total + 2$ then
 8:         $outIndex \leftarrow$ atomicAdd($outWLSize, max - min + 1$)
 9:         for $i = \text{min}; i <= max; i++$ do
10:             setNumOfKnownChoices($outWL, outIndex, total + 1$)
11:             setNextChoiceIndex(outWL, outIndex, 2)
12:             copyFromTo(inWL, outWL, outIndex)
13:             setLastChoiceValue(outWL, outIndex, i)
14:             $outIndex \leftarrow outIndex + 1$
15:         end for
16:         Exit
17:     else
18:         setNextChoiceIndex($inWL, tid, index + 1$)
19:         return getChoiceValue($inWL, tid, index + 1$)
20:     end if
21: end function
```

Fig. 9. Algorithm behind the _choice function as implemented in ReexeStrategy. This function is executed on the device.

saved in the global space and thus are accessible by all threads. Next (line 5), the thread obtains the total number of choice values contained in the task assigned to it, and the number of `_choice` calls that it has seen so far on the current path (line 6). These two values, which are assigned to `total` and `index` respectively, are needed to determine whether the value of the current `_choice` call already exists in the current task. The algorithm checks `index >= total + 2` to determine if the

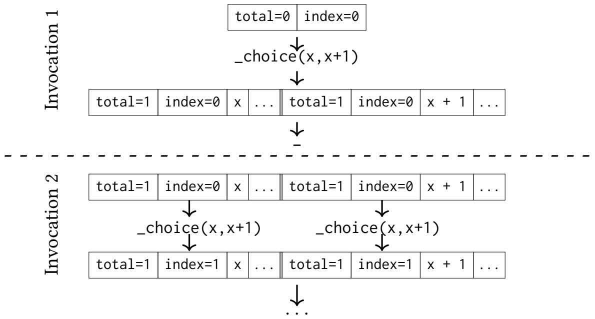


Fig. 10. Illustration of encoding tasks and updating worklists.

value for the current choice is known (line 7). Since the first two elements of a task are `index` and `total`, `index` is initially 2, so we add 2 to `total`. Based on that, there are two possible outcomes.

One, if the value is not known (true branch), the thread enters a loop (lines 9-15) to create a number of new tasks. First (line 8), it atomically obtains a location for the new tasks in the output worklist. Then, the thread increments the number of known choices (line 10), sets the choice index to its initial value (line 11), copies all choices from the current task (line 12), and sets the last choice in the new task based on the current iteration of the loop (line 13). Once the loop is done, the current thread terminates; if this was the only thread running, then Tempo takes over and schedules the next kernel call as already described in Figure 8. We visualize an example execution and the encoding in Figure 10. The upper part of the figure shows an execution that initially had no choices and then executed a single choice, creating 2 new tasks in the output worklist. The kernel then terminates, and the new tasks in the output worklist are eventually moved to the input worklist.

Two, if the value of the current choice *is* known (`!(index >= total+2)`), the thread increments `index` and returns the known value (lines 17-19). Execution then resumes normally. We illustrate this execution, i.e., when some choices are known, in the bottom part of Figure 10.

3.5 Scaling The Strategies

Ultimately both ForkStrategy and ReexeStrategy will hit a resource limit on GPUs when the number of paths grows large, either because of a lack of available threads (e.g., for ForkStrategy) or because of memory limitations (e.g., GPU memory capacities are typically limited). In this section, we describe extensions of the basic algorithms to enable exploration of programs with many paths.

3.5.1 ForkStrategy. While ForkStrategy stores only lightweight metadata and thus consumes significantly less memory than ReexeStrategy, ForkStrategy has to be run with a sufficient number of threads to ensure that the entire space is covered (Section 3.4.1). If the estimated number of threads is less than what the GPU supports, then we can simply use the estimated number. However, if the number exceeds the limit of the GPU, the runs would fail. In order to scale ForkStrategy, we extend the original algorithm to invoke the kernel multiple times, such that each launch explores a part of the test generation program. We highlighted this extension in figures 5 and 6.

In order to determine how many times to invoke the kernel, we specify how many threads to run per invocation: `chunk_size`. Dividing `ethreads` by `chunk_size` yields the number of kernel invocations needed, which we also call *chunks*. Two other modifications to ForkStrategy are needed. The goal is to make it seem that the GPU has `ethreads` threads running, when in reality that number is equal to `chunk_size`. First, the size of the initial, single task group is set to `ethreads`, instead of `chunk_size`. Thus, it appears like the GPU is running `ethreads` threads at the same time. Second, when a thread calls `_choice`, an offset is added to its `tid`. This offset is equal to `iteration_count` × `chunk_size`. `iteration_count` is the current chunk running on the GPU.

Since there are no dependencies between invocations of different chunks, they can potentially be executed on multiple GPUs concurrently, increasing the potential of parallelization in ForkStrategy.

3.5.2 ReexeStrategy. This strategy heavily uses memory to store tasks, limiting the maximum size of the input worklist. To enable exploration of larger spaces, we extend ReexeStrategy to offload tasks that cannot fit into the input worklist to main memory; this is similar to the approach used by Celik et al. [2017], but we integrate the offloading into the runtime so the user is unaware of its existence. In Figure 8, we highlight this extension (lines 12-13). Specifically, once a kernel launch finishes execution, we first move as many tasks as can fit from the output into the input worklist. Any additional tasks left in the output worklist are moved to the *overflow buffer*, which resides in the main memory. Next, if the input worklist is not full, we check if there are any outstanding items in the overflow buffer and move as many as we can from the overflow buffer to input worklist. This allows us to support arbitrarily large number of execution paths assuming sufficient main memory, at the cost of having to transfer the items between GPU and main memories.

3.6 A Brief Comparison

ForkStrategy and ReexeStrategy have different strengths and weaknesses. In this section we compare and contrast the two on an abstract level and we compare them empirically in Section 4.

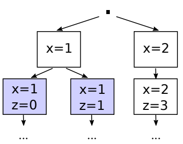
ReexeStrategy runs exactly as many threads as it needs, meaning that it utilizes the GPU's resources more efficiently than ForkStrategy, which could potentially run redundant threads, even with an estimate for the number of threads. There is also the possibility that the estimate is inaccurate, resulting in performance penalties if it is an overestimate, and execution failure if it is an underestimate. (Solving a failure due to underestimation is not hard, as we can simply start a new run and double the number of estimated threads until the run is successful, although we would have to pay the cost for non-successful runs.)

ForkStrategy only maintains two variables per thread, index and task group size. ReexeStrategy needs to store the value of every possible call to `_choice`. This means that ReexeStrategy runs fewer threads concurrently due to the limited memory available on a GPU. Additionally, at every kernel invocation, these tasks need to be transferred between different worklists and potentially even main memory. Moreover, ForkStrategy immediately chooses a value when `_choice` is called, unlike ReexeStrategy which has to stop execution when a new `_choice` is called and invoke the kernel again. This means that ReexeStrategy has to keep re-executing the same code over and over again. ForkStrategy, on the other hand, only re-executes the same code if the size of the search space is large enough to require multiple kernel launches. Finally, ReexeStrategy has to move tasks to and from the overflow buffer when running on a GPU, resulting in extra overhead.

Intuitively, we expect ForkStrategy to outperform ReexeStrategy, seeing as it immediately returns a value for `_choice`, instead of writing new tasks to memory, copying them back to the input worklist, moving to and from the overflow buffer, and then re-launching the kernel. In practice, however, ForkStrategy frequently overestimates the number of threads required, resulting in large numbers of redundant threads that do not contribute to exploration, especially for hybrid test generation programs which have irregular search spaces (Figure 1a).

3.7 MixedStrategy

MixedStrategy combines ForkStrategy and ReexeStrategy to get the best of both. At a high level, MixedStrategy first uses ReexeStrategy to build a *skeleton* of the search space, and then uses the skeleton to explore the search space with ForkStrategy. This skeleton contains information for every unique path in the test generation program. Each path contains all the choices on that path, as well as the number of threads that execute that path.



(a)

```
1 // control–flow _choice
2 x = _choiceCF(1, 2);
3 // regular _choice
4 z = (x == 1) ? _choice(0, 1) : 3;
5
6 _countIf(x <= 3 && z == 3);
```

(b)

Fig. 11. An example of a control flow choice.

The key insight behind the skeleton is that only some calls to _choice contribute to the irregularity of the search space. We call these *control flow* choices. Recognizing these calls is important during the skeleton building phase, so MixedStrategy extends the Tempo API to include a function for control flow choices: _choiceCF. The user is responsible for manually adding this function invocation where necessary. During skeleton building, only calls to _choiceCF will spawn new tasks. Figure 11b illustrates a control flow choice using a simple example. Figure 11a shows the search space during exploration.

ReexeStrategy. MixedStrategy uses ReexeStrategy to build the skeleton. It modifies the tasks (introduced in Section 3.4.2) to also include the number of threads needed for the current path, as well as a boolean value to indicate whether the task represents a complete path. Initially, the number of threads needed is one and all tasks in the input worklist are marked as complete. At a call to a new control flow _choiceCF (line 2 in Figure 11b), MixedStrategy creates new tasks, marks the current task in the input worklist as incomplete (since it spawns longer paths), and re-executes the kernel. At regular _choice calls (line 4 in Figure 11b), it updates the number of threads in tasks by multiplying it by the number of choices, and returns any value in the range. In Figure 11a, this means that only one of the colored nodes exists during skeleton building. Each complete task represents a unique path and is transferred to the host. Any tasks remaining in the output worklist are copied to the input worklist and the kernel is then re-executed.

ForkStrategy. Given the skeleton, MixedStrategy calculates the total number of threads needed for ForkStrategy by summing the number of threads in each unique path. It then initializes the thread metadata by creating one task group per unique path (in contrast to ForkStrategy which initially places all threads in one task group). MixedStrategy also needs extra metadata to retrieve control flow choices from the skeleton: the index of the path a thread belongs to, and the index of the next control flow choice in that path. During exploration, control flow _choice values are selected by indexing into the path's control flow choices (first level in Figure 11a), and task groups are divided in the same way as in ForkStrategy at regular _choice calls (second level in Figure 11a).

4 EVALUATION

In this section, we assess the performance of Tempo. We answer the following research questions:

RQ1. How does the performance of ForkStrategy compare to ReexeStrategy for hybrid test generation programs?

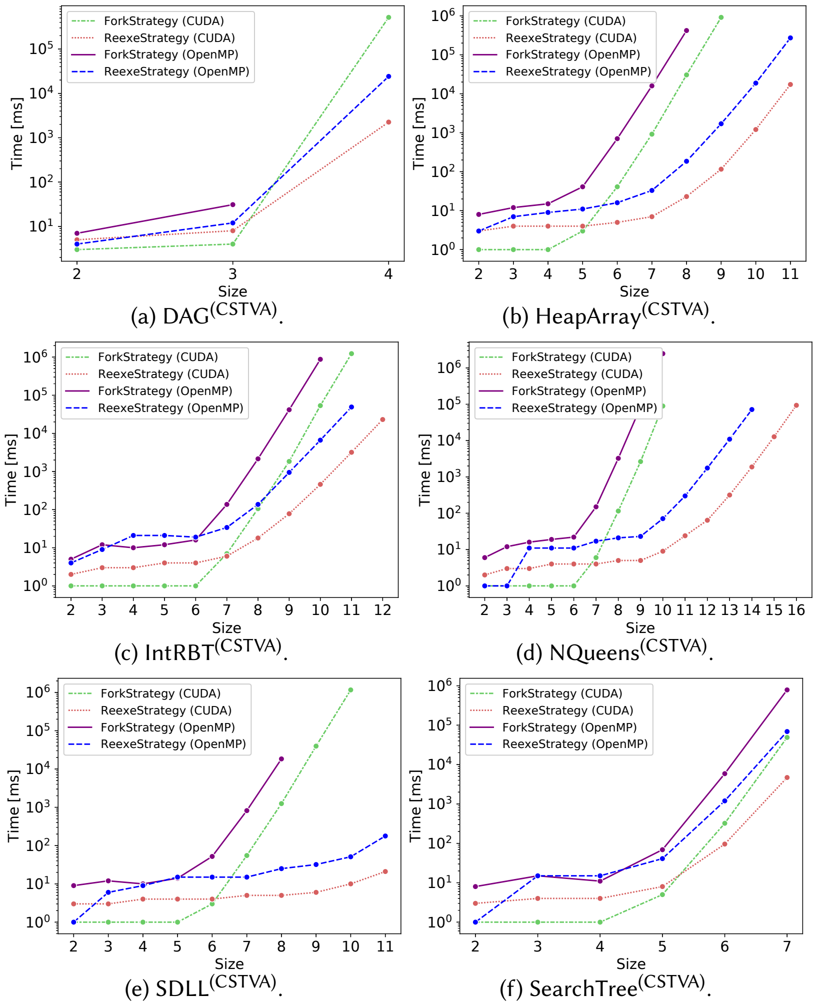
RQ2. How well does MixedStrategy improve on ForkStrategy and ReexeStrategy for hybrid test generation programs?

RQ3. How does Tempo compare to prior work on hybrid test generation programs?

RQ4. How does the performance of ForkStrategy compare to ReexeStrategy for sequence-based test generation programs?

RQ5. What are the values for thread divergence, achieved occupancy, and memory efficiency for Tempo's GPU backends?

Hardware and software configuration. We ran all experiments on a machine with 6-core Intel Core i7-8700 3.20GHz, and 64GB RAM. The same machine comes with an Nvidia GeForce RTX 2080 GPU with 8GB RAM. We used CUDA 11.0 and NVCC 11.0 for the GPU backend, and OpenMP 4.5 for the CPU backend.



(a) DAG$^{\text{(CSTVA)}}$.

(b) HeapArray$^{\text{(CSTVA)}}$.

(c) IntRBT$^{\text{(CSTVA)}}$.

(d) NQueens$^{\text{(CSTVA)}}$.

(e) SDLL$^{\text{(CSTVA)}}$.

(f) SearchTree$^{\text{(CSTVA)}}$.

Fig. 12. Size (x-axis) vs. Generation Time (y-axis) for Hybrid Test Generation Programs.

Note that our goal is not to compare CPU and GPU runs to establish which is better overall, but rather to characterize the tradeoffs for strategies and platforms to enable a system to dynamically choose among them.

We first introduce the subjects used in our study (Section 4.1). Then, we answer each research question (Sections 4.2-4.6). We include some additional results in the appendices.

4.1 Subjects

We evaluated Tempo by using 20 hybrid and sequence-based test generation programs (*generators* for short) for data structures which have been used in previous test generation studies [Celik et al. 2019, 2017; Galeotti et al. 2010; Gligoric et al. 2010; Kuraj et al. 2015; Sharma et al. 2011, 2010; Visser et al. 2006]. The names of all subjects used in our evaluation can be seen in the captions of the plots in Figure 12 and in the first column of Table 6; for each subject we indicate the original source (if applicable): CSTVA [Sharma et al. 2010], ISSTA [Visser et al. 2006], TACO [Galeotti et al. 2010].

Table 2. Results for Hybrid Test Generation Programs with MixedStrategy, ForkStrategy, and ReexeStrategy using CUDA.

Subject	**Size**	**Time [ms]**				
		MixedStrategy			**ForkStrategy**	**ReexeStrategy**
		Skeleton	Explore	Total	Total	Total
HeapArray$^{\text{(CSTVA)}}$	10	24	891	**915**	T/O	1208
	11	61	13936	**13997**	T/O	17314
IntRBT$^{\text{(CSTVA)}}$	11	23	1487	**1510**	1232926	3187
	12	60	12313	**12373**	T/O	23073
SearchTree$^{\text{(CSTVA)}}$	6	9	68	**77**	323	96
	7	10	3772	**3782**	48790	4690

4.2 ForkStrategy vs. ReexeStrategy Performance

In this section we compare ForkStrategy with ReexeStrategy for hybrid generation. For GPU exploration, we set the maximum number of threads (i.e., chunk size) per kernel invocation in ForkStrategy to 500 million. During thread estimation, the number of threads was 10k. The input worklist size chosen for ReexeStrategy was 40,960. For CPU exploration, we used 12 for threads per invocation and worklist size as preliminary experiments showed that this leads to the best results. The largest size/length that we used for each subject was set such that the run does not exceed one hour (and does not run out of memory). The generation times shown are the averages of three runs. All execution times were within 6% of the average for all generators.

Figure 12 shows the results for hybrid test generation programs. Each plot shows generation time (y-axis) for various sizes of the structures (x-axis), e.g., number of nodes in a red-black-tree.

RQ1. *How does the performance of ForkStrategy compare to ReexeStrategy for hybrid test generation programs?*

We can see in Figure 12 that ForkStrategy outperforms ReexeStrategy for smaller sizes but not for larger sizes for CUDA. For OpenMP, ReexeStrategy outperforms ForkStrategy across most sizes, except for some of the smaller sizes in IntRBT$^{\text{(CSTVA)}}$, SDLL$^{\text{(CSTVA)}}$, and SearchTree$^{\text{(CSTVA)}}$.

Comparing the total number of structures generated to the number of estimated threads in ForkStrategy (Table 9 in Appendix B), we find that the actual number of threads needed was much less than the estimated number. The difference between the two values increases as structure size increases, meaning that larger sizes are disproportionately affected by overestimation in comparison to smaller sizes. Thus, ForkStrategy is slower for hybrid generators as size increases. ReexeStrategy, meanwhile, performs better for larger sizes as it only ever runs exactly as many threads as it needs, meaning that it scales better than ForkStrategy as the number of paths grows.

In summary, we recommend ReexeStrategy for complex generators that have hard-to-estimate number of paths. In the future, we plan to encapsulate the choice of strategy into our runtime based on characteristics of a given test generation program, e.g., cyclomatic complexity.

4.3 MixedStrategy

RQ2. *How well does MixedStrategy improve on ForkStrategy and ReexeStrategy for hybrid test generation programs?*

Table 2 compares the CUDA implementations of MixedStrategy, ForkStrategy, and ReexeStrategy. Column 1 shows the name of the subject. We show the three subjects where MixedStrategy is applicable: HeapArray$^{\text{(CSTVA)}}$, IntRBT$^{\text{(CSTVA)}}$, and SearchTree$^{\text{(CSTVA)}}$. Column 2 shows the size. For MixedStrategy, we show skeleton building time, exploration time, and total time. For ForkStrategy and ReexeStrategy, we show total time.

MixedStrategy works best when a generator has fewer control flow choices than regular choices, as this leads to a smaller skeleton which can be built and explored quickly.

The data shows that MixedStrategy outperforms both ForkStrategy and ReexeStrategy. MixedStrategy is faster than ForkStrategy as the latter greatly overestimates the number of threads required, while the former produces an exact number. While MixedStrategy executes the same number of threads as ReexeStrategy, it does so over fewer kernel launches with fewer transfers to and from the overflow buffer, thereby improving performance.

Table 3. Test Generation Programs LOC and Character Count Comparison between Tempo and BET Tools.

Subject	**Tempo**		**UDITA**		**Korat^o**		**Korat^j**		**Korat^c**		**Korat^g**	
	LOC	Chars	LOC	Chars	LOC	Chars	LOC	Chars	LOC	Chars	LOC	Chars
HeapArray$^{(CSTVA)}$	19	568	19	454	34	893	47	1399	55	1520	243	7884
IntRBT$^{(CSTVA)}$	71	2277	62	1482	124	3266	222	5898	239	6247	661	18211

4.4 Comparison with Prior Work

RQ3. *How does Tempo compare to prior work on hybrid test generation programs?*

UDITA [Gligoric et al. 2010; UDITAWebPage [n. d.]] is the only available tool for exploring hybrid test generation programs. It explores the search space by backtracking to `_choice` calls and picking different values each time, in contrast to Tempo which uses parallelism and re-execution. Furthermore, UDITA implements lazy evaluation of `_choice` calls by delaying execution until the first use of the returned value. Korat^o [Boyapati et al. 2002] explores test generation programs written in a purely declarative style. Korat^j [Celik et al. 2017] is a variant of Korat^o with different encoding of the search space. Korat^c is an implementation of Korat^j for C, and Korat^g is a variant of Korat^c running on a GPU. We report numbers for two subjects that prior work has in common. Table 3 shows lines of code (LOC) and number of characters (Chars) for each test generation program. Table 4 shows generation time. Table 5 shows the number of valid structures found, as well as the total number of candidate structures explored using the hybrid generator (i.e. Tempo or UDITA) and the purely declarative generator (i.e., Korat).

Looking at LOC and Chars, we can see that generators written for Tempo and UDITA are more concise than declarative generators for other tools. Korat requires that the user explicitly specifies the bounds for each field of a subject. Additionally, Korat^j, Korat^c, and Korat^g encode objects as integer arrays to achieve better performance, i.e., they use no classes for data structures and field accesses are replaced with array accesses. This significantly increases LOC and code complexity.

As for generation time, we used MixedStrategy and ReexeStrategy with CUDA as they proved to be the fastest configuration for hybrid generators on our machine. Tempo is substantially faster than UDITA and scales better. Additionally, for HeapArray$^{(CSTVA)}$, Tempo is faster than all Korat implementations, and for IntRBT$^{(CSTVA)}$, Tempo is faster than all Korat implementations up to size 10, after which Korat^g scales better. By looking at Table 5, we can see that the total number of candidate structures explored, which depends on the style of the generator, correlates heavily with scalability. The hybrid HeapArray$^{(CSTVA)}$ generator explores less candidate structures than its pure declarative counterpart, so Tempo finishes exploration faster while still arriving at the same number of valid structures. This is in contrast to the hybrid IntRBT$^{(CSTVA)}$ generator, which explores more candidate structures than the declarative version. However, we still observe significant improvement over the sequential Korat implementations, Korat^o, Korat^j, and Korat^c. Other differences in performance between Tempo and Korat^g can be attributed to the latter benefiting from GPU specific optimizations used in their test subjects [Celik et al. 2017], such as encoding objects as integer arrays and using

Table 4. Test Generation Time Comparison between Tempo and BET Tools.

Subject	Size	Time [ms]						
		MixedStrategy	ReexeStrategy	UDITA	Korat^o	Korat^j	Korat^c	Korat^g
HeapArray$^{(CSTVA)}$	8	**22**	23	75180	755	463	322	49
	9	**102**	117	752720	5979	4716	3360	311
	10	**915**	1208	T/O	68135	54963	41218	3249
	11	**13997**	17314	T/O	936175	710350	527978	39845
IntRBT$^{(CSTVA)}$	9	**42**	79	18320	1595	947	649	62
	10	**221**	461	86450	5728	4770	3539	257
	11	1510	3187	409190	28217	26114	19941	**1297**
	12	12373	23073	1950230	150191	146292	112717	**7833**

Table 5. Structures Explored using Hybrid and Declarative generators.

Subject	Size	Valid Structures	Explored Structures	
			Hybrid	Declarative
HeapArray$^{(CSTVA)}$	8	1,005,075	**1,005,075**	5,231,385
	9	10,391,382	**10,391,382**	51,460,480
	10	111,511,015	**111,511,015**	583,317,405
	11	1,533,143,860	**1,533,143,860**	6,913,561,920
IntRBT$^{(CSTVA)}$	9	122	2,489,344	**1,510,006**
	10	260	17,199,104	**7,530,712**
	11	586	120,393,728	**39,089,158**
	12	1,296	852,017,152	**205,512,574**

bitset operators. We refrained from doing so for Tempo as we prioritized ease-of-use over other metrics.

Our findings are consistent with prior work [Gligoric et al. 2010]: hybrid test generation programs are more concise, but their efficient exploration is challenging. Tempo outperforms UDITA and also outperforms intKorat in most cases, bringing the performance of exploring hybrid test generation programs close to exploring non-hybrid programs with tools that limit developers' flexibility (long test generation programs purely written in declarative style).

4.5 Sequence-Based Generators

RQ4. *How does the performance of ForkStrategy compare to ReexeStrategy for sequence-based test generation programs?*

Table 6 shows the results for sequence-based test generation programs. Column 1 shows the name of the generator. Column 2 shows the length of the structure (e.g., length of the sequence of method calls). Column 3 shows the generation time for ForkStrategy (CUDA), ReexeStrategy (CUDA), ForkStrategy (OpenMP), and ReexeStrategy (OpenMP) respectively. Column 4 shows the number of times the kernel was invoked during GPU exploration.

The data shows that ForkStrategy outperforms ReexeStrategy for CUDA for almost all subjects. On the other hand, ReexeStrategy with OpenMP is faster than ForkStrategy.

For both CUDA and OpenMP, ForkStrategy was able to accurately estimate the number of threads required for every sequence-based subject (except for DisjSet$^{(CSTVA)}$) and for all lengths. This means that there were zero redundant threads run by ForkStrategy. This is expected as sequence-based generators follow a very simple pattern (Section 2). We intentionally picked sequence-based testing to observe the behavior of ForkStrategy in an (close to) ideal scenario with no redundant threads. We

Table 6. Results for Sequence-based Test Generation Programs with ForkStrategy and ReexeStrategy using CUDA and OpenMP.

Subject	**Length**	**Time [s]**				**#Kernels**
		CUDA		**OpenMP**		
		ForkStrategy	ReexeStrategy	ForkStrategy	ReexeStrategy	FS/RS
AVL$^{\text{(CSTVA)}}$	7	1	2	21	20	1/3178
	8	70	117	968	864	9/125842
AVL$^{\text{(TACO)}}$	7	1	2	21	19	1/3178
	8	65	111	950	844	9/125842
BinHeap$^{\text{(ISSTA)}}$	7	1	2	21	20	1/3178
	8	65	105	943	854	9/125842
BinTree$^{\text{(ISSTA)}}$	7	0	2	21	20	1/3178
	8	60	97	935	864	9/125842
CL$^{\text{(TACO)}}$	7	1	2	21	19	1/3178
	8	66	113	960	848	9/125842
DisjSet$^{\text{(CSTVA)}}$	5	2	0	54	2	1/378
	6	1705	34	T/O	529	279/75930
FibHeap$^{\text{(CSTVA)}}$	7	14	16	22	21	1/3178
	8	758	865	974	903	9/125842
FibHeap$^{\text{(ISSTA)}}$	7	1	2	22	20	1/3178
	8	63	103	962	876	9/125842
HeapArray$^{\text{(CSTVA)}}$	7	0	1	21	19	1/3178
	8	50	81	923	826	9/125842
IntRBT$^{\text{(CSTVA)}}$	7	1	2	21	20	1/3178
	8	62	104	955	863	9/125842
LL$^{\text{(TACO)}}$	7	1	2	21	20	1/3178
	8	62	109	930	843	9/125842
SLL$^{\text{(TACO)}}$	7	0	2	20	19	1/3178
	8	53	87	915	835	9/125842
TreeMap$^{\text{(ISSTA)}}$	7	1	2	21	20	1/3178
	8	64	103	951	850	9/125842
TreeSet$^{\text{(TACO)}}$	7	1	2	21	20	1/3178
	8	64	102	951	876	9/125842

checked DisjSet$^{\text{(CSTVA)}}$ in detail and found that, due to the code structure (an extra non-deterministic call in one of the branches), the estimation phase ended up being imprecise.

ReexeStrategy with CUDA moves a significant amount of tasks to the overflow buffer at almost every kernel launch, which causes the bulk of the observed overhead compared to ForkStrategy. As sequence length increases, the number of tasks and kernel launches increases, resulting in increased overhead. When running with OpenMP, threads write directly to an overflow buffer in main memory, so the overhead is almost negligible in comparison.

In summary, ForkStrategy should be used for GPU exploration, while ReexeStrategy is better for CPU exploration for sequence-based generators.

4.6 GPU Profiling Metrics

RQ5. *What are the values for thread divergence, occupancy, and memory efficiency for* Tempo*'s GPU backends?*

Table 7. Average Values for CUDA Profiling Metrics Across all Test Generation Programs.

Strategy	Generator Type	WEE [%]	AO [%]	GST [GB/s]	GLT [GB/s]
ForkStrategy	Sequence	26	88	24	32
ForkStrategy	Hybrid	23	92	14	22
ReexeStrategy	Sequence	26	84	103	204
ReexeStrategy	Hybrid	25	84	91	137

We profiled GPU exploration with CUDA by running all subjects using nv-nsight-cu-cli [nsightWebPage 2020] and obtained the following metrics:

- **Warp Execution Efficiency (WEE)** shows the effect of control flow divergence on Tempo. Control flow divergence occurs when threads within the same warp diverge into different execution paths. Larger values of WEE imply fewer control flow divergence.
- **Achieved Occupancy (AO)** shows the ratio of average active warps to maximum number of warps possible across streaming multiprocessors. Larger values of AO imply fewer cases of a warp stalling.
- **Global Store Throughput (GST)** shows the throughput for storing to GPU global memory.
- **Global Load Throughput (GLT)** shows the throughput for loading from GPU global memory.

Table 7 shows the results obtained by the profiler. Column 1 shows the runtime being profiled. Column 2 shows the type of the test generation program, i.e., either sequence-based or hybrid. Columns 3 to 6 show the averages of the profiling metrics we collected for WEE, AO, GST, and GLT. The values shown represent the average value of all generators; profiling results for ForkStrategy do not include the estimation phase. The length/size of each subject is the largest value for which the profiler does not run out of memory.

For ForkStrategy, WEE is 26% and 23% for sequence-based and hybrid generators. For ReexeStrategy, these numbers are 26% and 25%. These values are similar across both strategies, with sequence-based generators having slightly better WEE than hybrid generators. This indicates that differences in performance between the two strategies cannot be attributed to warp divergence.

For ForkStrategy, AO is 88% and 92% for sequence-based and hybrid generators. For ReexeStrategy, these numbers are 84% and 84%. Recall that achieved occupancy refers to the percentage of warps active in a kernel launch. The amount of warps active on a streaming multiprocessor is limited by the resources available, such as registers. GPUs rely on hardware multi-threading to exploit memory-level parallelism (MLP), context switching between warps on cache misses. Consequently, a warp can stall when it is waiting for the result of a memory access operation, and workloads with lower MLP can have lower occupancy as a result. Therefore, AO being larger for ForkStrategy is expected, as it is has a simpler program state compared to ReexeStrategy resulting in less register usage, and accesses memory less frequently.

5 A CASE STUDY

The main goal of this study was to evaluate our approach by experimenting with writing generators using Tempo. The purpose is to showcase the robustness of Tempo, although other test generation approaches, including UDITA and Korat, could have also been used. We generated several types of CUDA programs to be used as inputs to the Nvidia CUDA compiler [nvccWebPage 2019] (NVCC) and Clang [clangWebPage 2020], which can also be used to compile CUDA programs [clangCUDAWebPage 2020]. We also generated the corresponding C/C++ programs to test GCC [gccWebPage 2020] and Clang.

5.1 Program Types

We briefly describe several types of programs we aimed to generate.

Call graphs. We modified the DAG generator to generate cyclic graphs that represent call graphs. Each node of the generated graphs is a function, and a directed edge indicates a function call.

Inheritance graphs. Since valid inheritance graphs are essentially DAGs, the inheritance graph generator is almost identical to the DAG generator. The only difference between the two is that the inheritance graph generator does not include a NULL node, while the DAG generator does. Each node of the inheritance graph maps to a specific class. A directed edge indicates an inheritance relationship. There is also a choice that selects either a method (pure virtual, virtual, or neither) or a member variable (static or non-static) per class. Additionally, we built on top of the original inheritance graph generator to generate more specialized graphs:

- **Diamond inheritance graphs**: C++ allows diamond-shaped inheritance hierarchies, where a class inherits from two parent classes, and both parent classes inherit from the same parent class. This generator expands inheritance graphs by filtering graphs that do not contain a diamond.
- **Templated inheritance graphs**: Builds on top of the diamond generator by adding template parameters to the classes.
- **Complex inheritance graphs**: Builds on top of the diamond generator by adding methods with varying return types and arguments, as well as varying the inheritance access specifier (e.g. `private`, `protected`, or `public`) and optionally using the `virtual` specifier. [1]

Nested classes. A nested class is a class that has been defined inside the body of another class. The generator for this case is simple: for structures of size n, n classes are generated. Each class includes a data member, and has a nested class defined in its body. Additionally, one of the classes will include a member function which will attempt to access a data member of another class.

Function specifiers. Two functions are generated, with one calling the other. The function making the call will have `n` specifiers. The possible specifiers are `static`, `extern`, `constexpr`, `inline`, `volatile`, and `noexcept`. The CUDA programs can also contain `__device__`, `__forceinline__`, and `__noinline__`. Additionally, for the generated CUDA programs, there is an option for the calling function's body to be wrapped by a macro guard that checks if the macro `__CUDA_ARCH__` is defined. A common use-case for this macro is when a function is defined as both `__device__` and `__host__`, and part of it is only meant to be executed when the function is called from the device.

5.2 Setup

We used differential testing [McKeeman 1998] as the test oracle, which has been commonly used in the past when testing compilers [Yang et al. 2011] and other program analysis tools (e.g., [Kapus and Cadar 2017]). We compile CUDA programs with NVCC and Clang, and C++ programs with GCC and Clang, investigating any difference in compiler *errors*. So far, we have *not* used the output of program executions as an additional oracle for successfully compiled programs; we leave this for future work. We tested GCC 7.4.0, Clang 11.0.0, and NVCC 10.2 in their default configurations.

5.3 Issues Found

We detected and reported four issues, two of which were confirmed as new bugs in NVCC, and two of which were known bugs in Clang. We show these issues in Figure 13. Note that the variable and function names have been modified from what was originally generated for presentation purposes.

Constexpr. The program in Figure 13a was generated by the "Function specifiers" generator. In this case, NVCC successfully compiles the program, although it should report an error like Clang does. The reason compilation is supposed to fail is because the `constexpr` function `f1()` might

[1] https://en.cppreference.com/w/cpp/language/derived_class

```
1  __device__ int x = 0;
2  __device__ int f2() {
3    x += 99999;
4    return x; }
5  __device__ constexpr int f1() {
6    #ifdef __CUDA_ARCH__
7    f2();
8    #endif
9    return 999999999; }
10 __global__ void kernel() {
11   f1(); }
```

(a) NVCC Constexpr bug

```
1  class c2 {
2  public:
3  __device__
4  virtual void n() = 0;
5  };
6  class c1 : public c2 {
7  public:
8  __device__ void n() {
9    printf("c1\n");}
10 };
11 __global__ void kernel() {
12   c1 o0;
13   o0.n();}
```

(b) Clang Virtual bug

```
1 class E {
2 public:
3   class I { public: int y; };
4   void m() { new I()->y = 77; }
5 };
```

(c) NVCC Field access bug

```
1 class c2 {
2   public:
3     static int v0;};
4 class c1 : public c2 { }
5 int c2::v0 = 2;
6 __global__ void kernel {
7   printf("%d\n", c2::v0);}
```

(d) Clang Static bug

Fig. 13. Issues detected by Tempo in CUDA compilers.

be evaluated at compile time, and based on the C++ standard this function is not allowed to use non-constant expressions or to invoke functions that are not `constexpr` [constexpr 2020]. However, in this program, `f1` calls a non-`constexpr` function `f2()`.

Virtual. The program in Figure 13b was generated by the "Inheritance graphs" generator. Clang fails to compile the program, although it should succeed like NVCC does. The error message reported by Clang is `ptxas fatal : Cannot take address of function '__cxa_pure_virtual'` from `ptxas` (the Nvidia assembler). The reason for this error is that Clang does not provide the standard runtime function `__cxa_pure_virtual()`, causing a linking error. Interestingly, at higher levels of optimization, the function call is removed by the compiler and compilation succeeds.

Field access. The program in Figure 13c was generated by the "Nested classes" generator. It compiles successfully with NVCC, but not with Clang. The cause of this bug is line 4, specifically `new I()->f`. The C++ standard states that the precedence of `->` is higher than `new` [operator precedence 2020], and thus the expression should be parsed as `new (I()->f)`, which is an invalid use of `new`. NVCC parses it as `(new I())->f`, and so compiles without errors.

Static. The program in Figure 13d was generated by the "Inheritance graphs" generator. Clang successfully compiles this program, while NVCC fails. Clang wrongly allows using a static variable in device code (Line 7). Newer versions of Clang include a fix for this issue.

6 LIMITATIONS AND FUTURE WORK

As with other bounded-exhaustive techniques, Tempo suffers from the exponential explosion. We envision Tempo being used to systematically test only a few features at a time; other techniques, e.g., fuzz testing [Zeller et al. 2019], can combine several systematically generated programs. We checked correctness of our code by checking the number of structures across strategies, as well as comparing the numbers of generated structures with the numbers reported in prior work. In our experiments, the estimated number of threads for ForkStrategy was precise or an over-approximation; we plan to develop a mechanism to dynamically rerun a test generation program with larger number of threads if the estimate is incorrect. We did not use dynamic parallelism (i.e., launching kernels from other kernels) as our initial experiment showed that it introduces substantial overhead.

One potential threat to our study is whether speeding up BET and hybrid test generation programs is even necessary. Since exhaustive test generation could lead to a large number of generated tests, the bottleneck might be test execution rather than generation. However, this is not necessarily the case, and it depends on generators and filters in use.

Table 8. Test Generation and Execution Time for GCC and NVCC Case Study.

Subject	Size	#Tests	Generation [ms]	Execution [ms]	
				GCC	NVCC
InheritanceGraphComplex	3	432	15	209	897
	4	22032	1694	1661	5378
InheritanceGraphDiamond	3	2	10	126	277
	4	52	421	229	1232
InheritanceGraphTemplate	3	32	9	367	2187
	4	1088	5306	691	3181
PointerGraph	2	5	7	118	275
	3	171	9	181	880

Table 8 shows test generation and execution time for some of the generators from the previous section. The generation time reported uses ReexeStrategy, while test execution was parallelized by passing 32 programs to the compiler simultaneously and running 6 instances of the compiler concurrently. It can be seen that the ratio of generation to execution time varies greatly by generator and by test subject, i.e., NVCC is noticeably slower than GCC. For the InheritanceGraphTemplate generator, test generation dominates execution with both compilers, as it was designed to explore a larger number of candidate structures when compared to the other generators, due to allowing a variable number of class template parameters as well as more parent classes per each class. In the future, we plan to further study optimizing both generation and execution by reasoning about both steps at once.

In our experiments, we used a CPU and GPU available on an off-the-shelf Dell machine. Our results and conclusions might differ on different hardware. To address this, we obtained access to a machine with 8-core Intel Core i9-9900K 3.60GHz, and 32GB RAM. Although the absolute numbers somewhat differ, our high-level findings still hold.

There are several future directions. First, we plan to develop an approach that automatically determines the worklist size for a given program and device for ReexeStrategy. Second, it would be worth studying how to select the appropriate device for exploration [Chikin et al. 2019]. Third, we plan on finding the optimal memory layout for our worklists [Franco et al. 2017; Majeti et al. 2016].

In our case study, we did not consider the effects of undefined behavior in our generated programs. In the future, we plan on experimenting with the idea of using filtering to eliminate some classes of undefined behavior (e.g., infinite recursion).

7 RELATED WORK

Automated test generation. Korat [Boyapati et al. 2002] uses formal specifications of inputs (inside an imperative language) to exhaustively generate test inputs up to a given size, and relies on backtracking and pruning to improve efficiency. ASTGen [Daniel et al. 2007] allows the user to write abstract syntax tree generators for Java programs. UDITA [Gligoric et al. 2010] is a DSL that allows the user to write hybrid test generation programs, and uses backtracking (as implemented in a modified bytecode interpreter) to explore the search space. HyTeK [Rosner et al. 2014] uses a set of invariants that are either declarative, imperative, or a hybrid. These invariants are passed to a SAT solver that generates satisfying test inputs. Iorek [Ringer et al. 2017] allows the programmer to express how inputs differ from each other and then uses an SMT solver to generate different instances of those inputs. Jarvis [Peleg et al. 2018] creates property-based tests from a set of pre-existing unit tests. SciFe [Kuraj et al. 2015] generates complex structures from Enumerator objects, which are defined as functions that map from natural numbers to structures, and are implemented in a domain specific language embedded within Scala. Constraint logic programming [Dewey et al. 2015a] can be used to specify complex data structures in a declarative style. This approach typically results in better performance than the hybrid style at the cost of usability. Crux [Tomb et al. 2020] uses symbolic execution to test a program exhaustively. Unlike prior work, we developed the first parallel execution model that can execute hybrid test generation programs.

Utilizing GPUs for testing. Yaneva et al. [2017] presented a tool that compiles C programs and (existing) test cases into code that runs on the GPU. This results in speedups compared to execution of tests on a CPU. Our focus is on parallel test input generation (not on test execution).

intKorat [Celik et al. 2017] is an approach for bounded exhaustive test case generation on the GPU that implements the Korat algorithm [Boyapati et al. 2002]. It introduced a new abstract representation of the generated test cases which is necessary for implementing Korat on the GPU, and has the added benefit of faster execution of the invariant checking even on CPU.

GVM [Celik et al. 2019] is a Java bytecode interpreter that runs on the GPU. It was used to speed up sequence-based test case generation and execution of randomly generated test cases; there were substantial changes to GVM necessary to enable parallel runs of sequence-based testing. TornadoVM [Clarkson et al. 2018] is a Java framework for writing applications for heterogeneous systems that include GPUs and CPUs. It uses JIT compilation to dynamically configure applications at run-time depending on the target device. Tempo is an execution model for hybrid test generation programs, which requires no user knowledge of the underlying hardware architecture.

Testing compilers. Using Tempo, we wrote bounded exhaustive test generation programs for CUDA compilers and found a couple of new bugs. Automated test generation for compilers is a widely studied research area [Chen et al. 2020]. Csmith [Yang et al. 2011] is a random test generation tool for C compilers. CLsmith [Lidbury et al. 2015] applies fuzz testing to OpenCL compilers. Constraint logic programming [Dewey et al. 2015b] has been applied to the Rust typechecker in combination with fuzzing to find bugs. Skeletal program enumeration [Zhang et al. 2017] involves enumerating a set of programs from a syntactic skeleton structure (with *holes* for variables) to exhaustively explore all possible variable usage patterns.

8 CONCLUSIONS

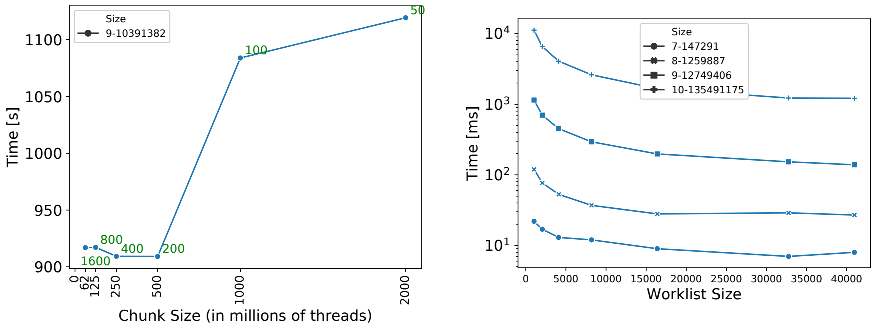
We presented novel programming and execution models for exploring hybrid test generation programs. We provide a runtime and two exploration strategies, both supported on GPUs and CPUs; this support comes without any extra burden on the users. Furthermore, the runtime automatically balances resource usage to enable exploration of test generation programs for large bounds. We used Tempo to generate instances of various data structures, which have been established as a standard benchmark in this domain and showed advantages over an existing framework for exploring hybrid test generation programs. Tempo requires no knowledge of the execution environment or parallelism and enables developers to efficiently explore their hybrid test generation programs without sacrificing expressiveness.

A IMPACT OF GLOBAL PARAMETERS ON ALGORITHMS

This section shows the impact of varying chunk size and input worklist size for ForkStrategy and ReexeStrategy respectively.

A.1 ForkStrategy

Figure 14a shows a plot of execution time vs. chunk size for the HeapArray$^{(CSTVA)}$ hybrid generator. The number next to each point indicates the number of kernel invocations for that size. In addition to showing the size, the legend shows the number of task groups for that size. As chunk size increases, execution time decreases until we reach size 500 million, after which there is a significant



(a) HeapArray$^{(CSTVA)}$ Hybrid Generator - Chunk size experiment.

(b) HeapArray$^{(CSTVA)}$ Hybrid Generator - Input worklist size experiment.

Fig. 14. Generation Time vs. Length/Size of Two Generators.

increase for sizes 1 billion and 2 billion, despite the decrease in the number of kernel invocations. The optimal value for chunk size seems to be 500 million for most hybrid generators, but the number varies for sequence-based generators. A larger chunk size means more threads running per kernel, all of which need to be scheduled to run on an SM. For extremely large numbers of threads, an SM might not always be available, so some threads would have to wait.

A.2 ReexeStrategy

For every generator, we measured the execution time for different structure sizes starting with an initial value of 1,024 for the input worklist size, and repeatedly doubled that value until we reached 32,768. We also include the value we used in our experiments from prior sections, which is 40,960.

The findings were consistent across all generators, both hybrid and sequence-based. For illustration purposes we only show the results of with the HeapArray$^{(CSTVA)}$ hybrid generator.

Figure 14b shows the plots of input worklist size vs. execution time for the HeapArray$^{(CSTVA)}$ hybrid generator for various structure sizes. In addition to showing the size, the legend shows the number of tasks created while generating the structure of that size. For most sizes shown, execution time is lowest when the size is 40,960. This is expected, as a larger input worklist means more tasks can be executed during a single kernel invocation, and therefore fewer kernel invocations are required, which leads to faster generation.

B HYBRID GENERATOR DATA

Tables 9 and 10 show more data collected from our Hybrid generators using ForkStrategy and ReexeStrategy with the CUDA backend, respectively. In both tables, Column 1 shows the name of the subject, Column 2 shows the size, Column 3 shows the number of valid structures generated, and Column 4 shows the number of kernel launches. In Table 9, Column 6 shows the number of estimated threads and Column 7 shows the total number of candidate structures explored.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments. We thank Ahmet Celik, Darko Marinov, Pengyu Nie, Zhiqiang Zang, Yuki Liu, Jiyang Zhang, Steven Zhu, Joseph Ryan, and Aleksandar Milicevic for their feedback on this work. This work was partially supported by a Google Research Scholar Award and the US National Science Foundation under Grant Nos. CCF-1652517, CCF-1704790, CCF-2107291, CNS-1846169, and CNS-2006943.

Table 9. Exploration Metrics for Hybrid Test Generation Programs with ForkStrategy using CUDA.

Subject	**Size**	**#Structs**	**Time [ms]**	**#Kernels**	**#EstimatedThreads**	**#TotalStructs**
DAG$^{(\text{CSTVA})}$	2	2	3	1	36	16
	3	34	4	1	82944	4802
	4	2352	517751	60	30000000000	100003691
HeapArray$^{(\text{CSTVA})}$	7	117562	918	1	134217728	117562
	8	1005075	30505	7	3486784401	1005075
	9	10391382	913576	200	100000000000	10391382
IntRBT$^{(\text{CSTVA})}$	9	122	1837	1	185794560	2489344
	10	260	53371	8	3715891200	17199104
	11	586	1232926	164	81749606400	120393728
NQueens$^{(\text{CSTVA})}$	8	92	115	1	16777216	13756
	9	352	2641	1	387420489	64337
	10	724	89203	20	10000000000	313336
SDLL$^{(\text{CSTVA})}$	8	12870	1249	1	150994944	12870
	9	48620	39622	8	3874204890	48620
	10	184756	1164363	220	110000000000	184756
SearchTree$^{(\text{CSTVA})}$	5	5292	5	1	375000	131250
	6	60984	323	1	33592320	6158592
	7	736164	48790	9	4150656720	353299947

Table 10. Exploration Metrics for Hybrid Test Generation Programs with ReexeStrategy using CUDA.

Subject	**Size**	**#Structs**	**Time [ms]**	**#Kernels**
DAG$^{(\text{CSTVA})}$	2	2	6	5
	3	34	8	10
	4	2352	2251	3075
HeapArray$^{(\text{CSTVA})}$	9	10391382	117	319
	10	111511015	1208	3314
	11	1533143860	17314	44095
IntRBT$^{(\text{CSTVA})}$	10	260	461	844
	11	586	3187	5838
	12	1296	23073	41247
NQueens$^{(\text{CSTVA})}$	14	365596	1879	9235
	15	2279184	12756	61845
	16	14772512	93434	440015
SDLL$^{(\text{CSTVA})}$	9	48620	6	11
	10	184756	10	16
	11	705432	22	35
SearchTree$^{(\text{CSTVA})}$	5	5292	8	13
	6	60984	96	196
	7	736164	4690	10417

REFERENCES

Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing locality and independence with logical regions. *International Conference on High Performance Computing, Networking, Storage and Analysis*, 1–11.

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

Ahmet Celik, Pengyu Nie, Christopher J. Rossbach, and Milos Gligoric. 2019. Design, Implementation, and Application of GPU-based Java Bytecode Interpreters. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 177:1–177:28.

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.

Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *ACM Computing Survey* 53, 1 (2020).

Artem Chikin, Jose Nelson Amaral, Karim Ali, and Ettore Tiotto. 2019. Toward an Analytical Performance Model to Select between GPU and CPU Execution. In *International Parallel and Distributed Processing Symposium Workshops*. 353–362.

clangCUDAWebPage 2020. NVCC. https://llvm.org/docs/CompileCudaWithLLVM.html.

clangWebPage 2020. Clang: a C language family frontend for LLVM. https://clang.llvm.org/.

James Clarkson, Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, Christos Kotselidis, and Mikel Luján. 2018. Exploiting High-performance Heterogeneous Hardware for Java Programs Using Graal. In *International Conference on Managed Languages & Runtimes*. 4:1–4:13.

constexpr 2020. constexpr specifier. https://en.cppreference.com/w/cpp/language/constexpr.

CUDADocs 2020. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

CUDAWebPage 2020. CUDA Zone. https://developer.nvidia.com/cuda-zone.

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

Kyle Dewey, Lawton Nichols, and Ben Hardekopf. 2015a. Automated Data Structure Generation: Refuting Common Wisdom. In *International Conference on Software Engineering*. 32–43.

Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015b. Fuzzing the Rust Typechecker Using CLP. In *Automated Software Engineering*. 482–493.

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

Juliana Franco, Martin Hagelin, Tobias Wrigstad, Sophia Drossopoulou, and Susan Eisenbach. 2017. You Can Have It All: Abstraction and Good Cache Performance. In *International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 148–167.

Juan Pablo Galeotti, Nicolás Rosner, Carlos Gustavo López Pombo, and Marcelo Fabian Frias. 2010. Analysis of Invariants for Efficient Bounded Verification. In *International Symposium on Software Testing and Analysis*. 25–36.

gccWebPage 2020. GCC, the GNU Compiler Collection. https://gcc.gnu.org/.

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

Daniel Jackson and Craig A. Damon. 1996. Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. In *International Symposium on Software Testing and Analysis*. 239–249.

Timotej Kapus and Cristian Cadar. 2017. Automatic Testing of Symbolic Execution Engines via Program Generation and Differential Testing. In *Automated Software Engineering*. 590–600.

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.

Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-Core Compiler Fuzzing. In *Conference on Programming Language Design and Implementation*. 65–76.

Deepak Majeti, Kuldeep S. Meel, Rajkishore Barik, and Vivek Sarkar. 2016. Automatic Data Layout Generation and Kernel Mapping for CPU+GPU Architectures. In *Proceedings of the 25th International Conference on Compiler Construction*. 240–250.

William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107.

José Meseguer and Grigore Roşu. 2013. The Rewriting Logic Semantics Project: A Progress Report. *Information and Computation* 231 (2013), 38–69.

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

nsightWebPage 2020. Nsight Compute CLI. https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html.

nvccWebPage 2019. NVCC. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html.

OpenMPIWebPage 2020. OpenMPI - A High Performance Message Passing Library. https://www.open-mpi.org.

OpenMPWebPage 2020. OpenMP. https://www.openmp.org.

operator precedence 2020. C++ Operator Precedence. https://en.cppreference.com/w/cpp/language/operator_precedence.

Hila Peleg, Dan Rasin, and Eran Yahav. 2018. Generating Tests by Example. In *Verification, Model Checking, and Abstract Interpretation*. 406–429.

Talia Ringer, Dan Grossman, Daniel Schwartz-Narbonne, and Serdar Tasiran. 2017. A Solver-Aided Language for Test Input Generation. *Proc. ACM Program. Lang.* 1, Conference on Object-Oriented Programming, Systems, Languages, and Applications.

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

Marija Selakovic, Michael Pradel, Rezwana Karim, and Frank Tip. 2018. Test Generation for Higher-Order Functions in Dynamic Languages. *Proc. ACM Program. Lang.* 2, Conference on Object-Oriented Programming, Systems, Languages, and Applications (2018).

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

Rohan Sharma, Milos Gligoric, Vilas Jagannath, and Darko Marinov. 2010. A comparison of constraint-based and sequence-based generation of complex input data structures. In *Workshop on Constraints in Software Testing, Verification and Analysis*. 337–342.

Aaron Tomb, Stuart Pernsteiner, and Mike Dodds. 2020. Symbolic Testing for C and Rust. In *2020 IEEE Secure Development (SecDev)*. 33–33.

UDITAWebPage [n. d.]. UDITA Home Page. http://mir.cs.illinois.edu/udita.

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.

Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. 2005. Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 365–381.

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.

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

Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. The Fuzzing Book. Saarland University.

Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal Program Enumeration for Rigorous Compiler Testing. In *Conference on Programming Language Design and Implementation*. 347–361.