The Indigo Program-Verification Microbenchmark Suite of Irregular Parallel Code Patterns

Yiqian Liu
Department of Computer Science
Texas State University
San Marcos, TX, US
y_l120@txstate.edu

Noushin Azami
Department of Computer Science
Texas State University
San Marcos, TX, US
noushin.azami@txstate.edu

Corbin Walters
Department of Computer Science
Texas State University
San Marcos, TX, US
ckw79@txstate.edu

Martin Burtscher
Department of Computer Science
Texas State University
San Marcos, TX, US
burtscher@txstate.edu

***Abstract*—Irregular programs are found in many domains and tend to exhibit input-dependent control flow and memory accesses. This paper introduces the Indigo suite of important irregular parallel code patterns for testing verification and other tools. We studied many irregular CPU and GPU programs and extracted the key code patterns. Then, we methodically built variations of these patterns to alter the control-flow and memory-access behavior and/or introduce bugs, yielding the thousands of OpenMP and CUDA microbenchmarks in the suite. Indigo includes a set of generators to systematically create an unbounded number of inputs for each microbenchmark, which is essential to exercise the wide range of possible behaviors of input-dependent codes. To manage the millions of code and input combinations, Indigo provides the flexibility to generate user-defined subsets of the suite. Experiments with a subset of buggy and bug-free codes illustrate that irregular programs pose a significant challenge to both static and dynamic program verification tools. Moreover, such tools can perform quite differently across code patterns that contain the same bug.**

***Index Terms*—benchmark design, parallel computing, irregular programs, software verification**

I. Introduction

Many computational problems are irregular in nature, meaning their control flow or memory accesses do not follow simple patterns. This irregularity often arises from processing pointer-based data structures like graphs that are used to represent real-world objects such as road networks. Irregular algorithms can be found in many domains, including social networking [1], data mining [2], artificial intelligence [3], and compilers [4]. Since many important applications are irregular, it is crucial to understand, support, and exploit the behaviors of these codes.

Every serial and parallel program has a degree of control-flow and memory-access irregularity [5]. Control-flow irregularity typically stems from *while* loops whose iteration count is difficult to predict. Visiting the neighbors of graph vertices is an example as each vertex may have a different number of neighbors. In contrast, regular codes like dense matrix multiplication tend to be based on *for* loops with fixed iteration counts. Memory-access irregularity typically stems from *pointer-chasing* operations where the address of the next access is difficult to predict. Again, visiting a vertex's neighbors is an example because the neighbors are rarely located in consecutive memory locations. In contrast, regular codes tend to perform *strided* memory accesses. For instance, the elements of a vector reside at consecutive memory addresses.

Since the control-flow and memory-access patterns of irregular code are generally input dependent and tend to change during program execution, the observed behavior for one input or time slice may not be representative of the behavior of the same code for a different input or time slice [5], making bug detection more difficult. Parallelism often exacerbates the problem as the relative timing behavior of the threads may change from run to run. Their data dependency can make irregular programs more challenging to analyze, optimize, verify, and parallelize than regular codes.

The growing importance of irregular applications is reflected in recent benchmark suites. Whereas many of the suites listed in Table I contain breath-first search (an irregular code), the older suites mostly consist of regular codes. Only some of the more recent suites focus on irregular programs. *However, all of them comprise just a handful of codes and inputs in total since they are performance benchmark suites. Thus, our community would greatly benefit from a more extensive set of programs and inputs that exhibits the wide range of behaviors possible in irregular codes.* Such a suite could help programmers, tool developers, computer architects, and researchers design software and hardware that can better handle irregular programs.

To drive the design of program verification tools, for which irregular codes are particularly challenging, it is important to also include common bugs. Except for DataRaceBench, such defective codes are absent from the benchmark suites in Table I, which lists the name, number of programs, release year, whether it is mostly irregular, and the parallel programming model of each suite.

This research was supported in part by the National Science Foundation under award No. 1955367 and by an equipment donation from NVIDIA Corp.

TABLE I
SELECTED BENCHMARK SUITES

Suite	Codes	Year	Irreg	Models
PARSEC [6]	12	2008	No	OMP, Pthreads, TBB
Lonestar [7]	22	2009	Yes	C++, CUDA
Rodinia [8]	23	2009	No	OMP, CUDA, OCL
SHOC [9]	25	2010	No	CUDA, OCL
Parboil [10]	11	2012	No	OMP, CUDA, OCL
PolyBench [11]	30	2012	No	CUDA, OCL
Pannotia [12]	13	2013	Yes	OCL
GAPBS [13]	6	2015	Yes	OMP
graphBIG [14]	12	2015	Yes	OMP, CUDA
Chai [15]	14	2017	No	AMP, CUDA, OCL
DataRaceBench [16]	168	2017	No	OMP, Fortran
GARDENIA [17]	9	2018	Yes	OMP (target), CUDA
GBBS [18]	20	2020	Yes	Ligra+

As a remedy, we designed Indigo, a parallel CPU and GPU suite of common irregular code patterns. Its goal is to provide the community with a *systematic* means to analyze irregularity in detail, expose potential bugs, study complex control-flow and memory-access behavior, and evaluate parallelization, optimization, and verification strategies on irregular programs.

To create Indigo, we studied the irregular codes in Lonestar and other suites, extracted the key patterns, generalized them, and methodically built variations thereof, including some with the types of bugs we have encountered when implementing our own irregular codes. The resulting codes are *microbenchmarks*, i.e., they are small, simple, and not full-fledged applications. For this reason, we do not recommend Indigo as a performance benchmark suite. However, being small is advantageous for static program analysis tools and cycle-accurate simulators that tend to be slow when analyzing or simulating large programs. In fact, all of the codes have a runtime that is linear in the number of vertices and edges. Version 0.9 of Indigo, upon which this paper is based, contains 1084 CUDA and 636 OpenMP microbenchmarks, including 628 CUDA and 324 OpenMP codes with bugs.

Due to the input-dependent nature of irregular codes, it is essential to also provide many different inputs (graphs). Rather than including predetermined inputs, Indigo comes with a set of graph generators that allow the user to create an unbounded number of inputs. To support systematic and exhaustive testing of the microbenchmarks, one generator emits all possible directed and/or undirected graphs with a user-specified number of vertices. Additionally, Indigo includes generators for power-law graphs [19], k-dimensional grids and tori [20], uniform distribution graphs [21], etc.

Each microbenchmark can be run with all generated inputs to elicit a wide variety of runtime behaviors. However, running all 1720 microbenchmarks on just the 4096 possible directed 4-vertex graphs[1] would result in 7,045,120 tests, which is probably too many for most use cases. After all, assuming each test takes one second, it would take close to three months to run the entire suite. Moreover, some users may not be interested in the buggy codes, others may only care about the CUDA programs, and yet others may want to study undirected graphs exclusively. To facilitate these and other use cases, Indigo generates not only the inputs but also the desired microbenchmarks based on a simple configuration file. This file can be edited to enable or disable various filters, thus allowing users to create any wanted subset of the suite. Indigo includes sample configuration files to build various subsets.

This paper makes the following main contributions.

- It presents the Indigo suite with 1720 input-dependent CUDA and OpenMP codes as well as an unbounded number of inputs for each code.
- It introduces a new type of benchmark suite that generates desired program variations and inputs on the user side.
- It describes six fundamental dwarf-like code patterns that frequently occur in parallel graph applications.
- It explains how Indigo methodically generates variations of code patterns, including planting bugs in them.
- It illustrates, based on hundreds of thousands of experiments, that irregular codes pose a significant challenge to many program verification tools.

The Indigo suite is available in open source at https://cs.txstate.edu/~burtscher/research/IndigoSuite/.

The rest of this paper is organized as follows. Section II reviews relevant background information. Section III summarizes related work. Section IV describes the design of the Indigo suite in detail. Section V discusses the experimental methodology. Section VI evaluates several CPU and GPU program verification tools on buggy and bug-free Indigo codes. Section VII summarizes the paper and draws conclusions.

II. BACKGROUND

This section provides background information on the used graph format and presents an irregular code example.

A. CSR Graph Format

The Compressed Sparse Row (CSR) format is one of the most widely used graph representations [22]. For example, Pannotia [12] and Lonestar [7] use CSR inputs. All Indigo graph generators produce graphs in this format, meaning that every generated graph can be used as an input for any Indigo code. Basing Indigo on the CSR format makes it easy for users to import their own graphs and means that preexising and real-world (non-synthetic) graphs can also be used as inputs.

B. Irregular Code Example

Determining the connected components (CCs) of a directed graph $G(V, E)$ is an important computation that can be implemented in different ways. One way is through push-style label propagation as outlined in Algorithm 1. First, the label of each vertex, $label$, is made unique by initializing it to the vertex ID (lines 1 to 3). Then, for each vertex v (line 7), all neighbors in the adjacency list adj are visited (line 8) and processed. The processing (lines 9 to 12) updates each neighbor's label with the label of v if v's label is larger. Whenever a label is updated, the flag $updated$ is set (line 11). The algorithm iterates until no more updates occur (line 5). Upon termination, all vertices in

[1]Note that we may not want to eliminate isomorphic graphs as vertex permutations result in different threads and warps processing a specific vertex.

the same CC will have the same label, and vertices in different CCs will have different labels.

Algorithm 1 Label-propagation-based connected components

Input: Graph $G = (V, E)$

```
1:  for all vertices v ∈ V do
2:      label[v] ← v
3:  end for
4:  updated ← true
5:  while updated do
6:      updated ← false
7:      for all vertices v ∈ V do
8:          for all neighbors n ∈ adj[v] do
9:              if label[n] < label[v] then
10:                 label[n] ← label[v]
11:                 updated ← true
12:             end if
13:         end for
14:     end for
15: end while
```

Output: Label of each vertex in G

Note that this label propagation algorithm is input dependent and has both control-flow (e.g., line 8) and memory-access (e.g., line 10) irregularity. It is impossible to statically predict the iteration count of the inner for loop without knowing the input graph. Similarly, it is impossible to statically predict the order in which the elements of the $label$ array will be written unless we know the complete input graph.

III. Related Work

Many benchmark suites exist. They target a plethora of different program behaviors, application domains, programming languages, etc. The early suites that focus on parallel programs mainly comprise *regular* high-performance computing (HPC) applications. One of the first regular suites not focusing on HPC is PARSEC [6], released in 2008. However, since the irregular benchmark suites published so far are performance rather than verification suites, none of them include enough inputs to elicit a wide range of distinct program behaviors.

DataRaceBench [16] is a relatively recent suite of regular programs designed to evaluate CPU data-race detection tools. It includes a set of kernels, some of which contain bugs. It comes with a script to evaluate Helgrind, Archer, ThreadSanitizer, Intel Inspector, and Coderrect Scanner. Verma et al. enhanced the suite by adding kernels that represent additional patterns and include FORTRAN code [23]. Program verification is also the target of Indigo, which supports OpenMP and CUDA, includes more bug types, inputs, and code versions, and provides customizable code and input generators.

With accelerators becoming popular, quite a few benchmark suites now include GPU code. The Rodinia [8] suite targets heterogeneous systems. It exhibits different types of parallelization, memory-access and data-communication patterns, synchronization, and power consumption. The SHOC [9] suite is designed to test the performance and stability of heterogeneous systems. Parboil [10] is a suite for evaluating the throughput of a range of applications, which can be used by programmers as a baseline to improve upon and/or for task-parallel programs. The Chai [15] suite evaluates the shared virtual memory, memory coherence, and system-wide atomics of heterogeneous systems as well as data- and task-based workload partitioning between the CPU and GPU.

There are several tools that target GPU program verification. GKLEE [24] searches for correctness and performance bugs in GPU codes. It includes 40 benchmarks that cover many CUDA program behaviors and issues such as thread divergence, bank conflicts, deadlock, and data races. GPUVerify [25] comes with a suite of 163 CUDA and OpenCL kernels drawn from public and commercial resources. Barracuda [26] is a concurrency bug detector for CUDA programs. It handles a wide range of parallelism constructs including branch operations, low-level atomics, and memory fences. It includes a concurrency bug suite with 53 programs, 12 of which have data races. Since essentially no verification suites with buggy GPU codes exists, all of these tools include their own.

The above mentioned benchmarks mostly contain regular programs. However, a growing number of suites focus on parallel irregular codes. Lonestar [7] contains C++ and CUDA implementations of iterative graph algorithms. Since it is the largest collection of irregular codes, we used it as the main source for extracting the irregularity patterns found in Indigo. Pannotia [12] is an OpenCL suite of applications for studying graph algorithms on GPUs. It includes 8 applications. GraphBIG [14] contains implementations of representative data structures, workloads, and data sets from 21 real-world use cases of multiple application domains. GAPBS [13] not only specifies graph kernels, input graphs, and evaluation methodologies but also provides optimized reference implementations. GARDENIA [17] is a benchmark suite for studying irregular graph algorithms on massively parallel accelerators. It includes 9 workloads from graph analytics, sparse linear algebra, and machine learning. GBBS [18] is a C++ suite of scalable, provably-efficient implementations of graph problems for shared-memory multicore machines. It extends the Ligra interface with additional primitives and clearly defined cost bounds. All of these benchmark suites include full-fledged graph codes. In contrast, Indigo comprises important code patterns that are not complete algorithms.

There are also benchmark suites for other parallel programming languages such as Go. Tu et al. analyzed the causes, detection, and fixes of 171 concurrency bugs from 6 popular Go software applications [27]. GoBench [28], the first suite for Go concurrency bugs, was introduced in 2021. It contains 82 real bugs from 9 open source applications and 103 bug kernels. It covers traditional and Go-specific concurrency issues. It uses configuration files in json format that record the type of bugs and describe how to generate the corresponding Docker files. Similarly, the configuration file used by Indigo defines the types of codes and inputs to be included in the generated suite.

The source code annotation and variation of CREST [29]

and DLBENCH [30] inspired the code generation process in Indigo. DLBENCH consists of a kernel generator, a profiler, and a performance analyzer to generate parameterized variants of a synthetic microbenchmark. CREST is a software framework that analyzes dependencies among GPU threads and performs source-level restructuring. It uses source-code annotations in the code restructurer to control optimizations.

In addition to focusing on common irregular code patterns, the main differences between Indigo and other benchmark suites are the much larger number of codes, the much higher number of inputs (which is important for data-dependent codes), and the support for creating user-defined subsets of the suite through configurable code and graph generators. Between the thousands of codes and the unbounded number of inputs, Indigo allows users to run millions of distinct tests and to create subsets for many different usage scenarios.

IV. Indigo Design

The primary goal of Indigo is to enable the systematic exploration of key parallel irregular code patterns. As mentioned, most existing suites do not focus on irregular programs. The few that do contain dozens of full-fledged graph kernels, each with just a few inputs, making them not particularly useful for systematic studies. Moreover, these suites do not contain buggy codes, making them unsuitable for program verification. Hence, we set out to create our own benchmark suite.

A. Graph Types

Since the behavior of irregular codes is data dependent, we may need a large number of inputs for each microbenchmark to elicit a wide variety of control-flow and memory-access-pattern combinations. Rather than providing a fixed set of inputs, we opted to include graph generators that allow the user to create any desired number of inputs. Importantly, one of the generators creates all possible directed and undirected graphs for a given number of vertices. The resulting graphs necessarily cover all corner cases that could appear in a real-world graph in this size range, making systematic and exhaustive testing possible. Since the number of possible graphs grows exponentially with the number of vertices, this generator cannot be used to create graphs with many vertices. Hence, we also included other generators to produce specific types of graphs with larger vertex counts. All generated inputs use the CSR format so that every microbenchmark can use all of them. Indigo includes the following graph generators.

- *All possible graphs*: this generator works by enumerating all possible adjacency matrices.
- *Binary forests*: this generator repeatedly picks a childless vertex and randomly assigns it an unvisited left child, right child, both, or none.
- *Binary trees*: this generator visits every vertex and randomly assigns it an unvisited left and/or right child.
- *Capped maximum-degree graphs*: this generator assigns up to k random edges to each vertex.
- *Directed acyclic graphs (DAGs)*: this generator assigns a random priority to each vertex and then creates random edges connecting higher- to lower-priority vertices.
- *k-dimensional grids*: this generator links each vertex to the next vertex in all dimensions.
- *k-dimensional tori*: this generator works like the grid generator but also connects the last vertex to the first vertex in all dimensions.
- *Power-law graphs*: this generator permutes the vertex list and then picks a source and destination vertex for each edge following a power-law distribution.
- *Random neighbor graphs*: this generator assigns a single random neighbor to each vertex.
- *Simple planar graphs*: this generator creates a random binary tree and links the internal nodes at the same level.
- *Star graphs*: this generator picks one random vertex and adds edges from that vertex to all other vertices.
- *Uniform-distribution graphs*: this generator is similar to the power-law generator but uses a uniform distribution.

Where applicable, the generators produce three versions of each graph: undirected, directed, and counter-directed (with the edge directions reversed). Figure 1 shows possible grids and tori that can be generated, and Figure 2 shows examples of the remaining supported graph types.

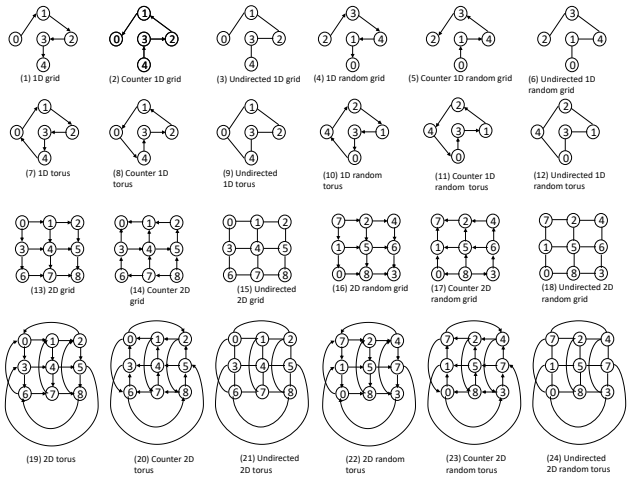

Fig. 1. Generated grid and torus inputs

Each generator takes a parameter that specifies the number of vertices. Some take a second parameter that specifies the maximum degree of the capped maximum-degree graph or the number of edges of the DAG, power-law, and uniform-distribution graphs. For the binary tree, torus, grid, random-neighbor, and star graphs, the number of edges is determined by the number of vertices. For the binary forests and the simple planar graphs, the number of edges is determined dynamically.

B. Major Code Patterns

As we are interested in common patterns of irregular codes, we conducted an extensive study of many irregular parallel C++ and CUDA programs, including programs from the

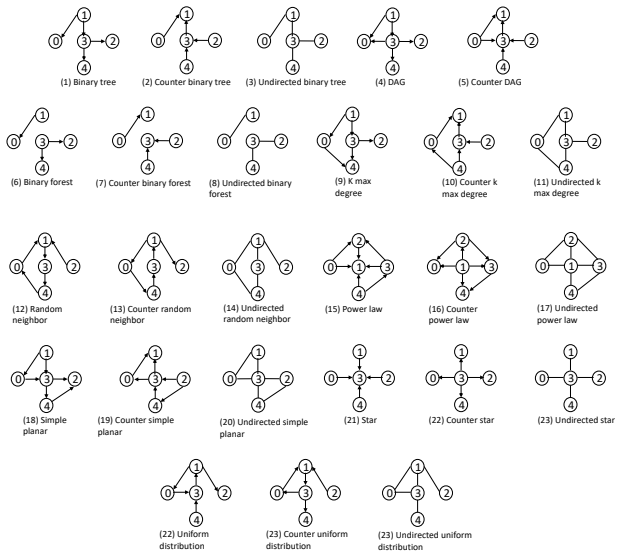

Fig. 2. Different types of generated input graphs

Lonestar and other suites. We then generalized the extracted patterns and narrowed them down to the following six major patterns. We gave them names to simplify the discussion.

- *Conditional-vertex pattern*: this code pattern updates a shared memory location if the neighbors of a vertex meet some condition. For example, in Lonestar, the k-clique and clustering codes read the neighbors' data (e.g., the cluster ID) and update a shared variable (e.g., the size of the cluster with the largest ID).
- *Conditional-edge pattern*: this code pattern updates a shared memory location if the edges of a vertex meet some condition. For example, in Lonestar, the triangle counting updates a global scalar if the edge is in an unexplored triangle, and the maximum cardinality bipartite matching adds the edge into a matching set if it does not share end points with any edges in the set.
- *Pull pattern*: this code pattern updates a vertex-private memory location based on some neighbors' data. E.g., graph coloring in Pannotia reads the neighbors' colors and SSSP in Lonestar reads the neighbors' distances.
- *Push pattern*: this code pattern updates a shared memory location in some neighbors based on vertex-private data. For example, page rank in Pannotia transfers the pagerank value to the neighbors, and the maximal independent set code in Lonestar marks the neighbors as 'out' of the set.
- *Populate-worklist pattern*: this code pattern conditionally places vertices (or edges) in unique but contiguous elements of a shared array. For example, BFS in Pannotia dynamically maintains a worklist of the vertices at the same level, and SSSP in Lonestar adds and removes vertices from the worklist depending on their distance.
- *Path-compression pattern*: this code pattern traverses partially shared paths and updates some vertices on the path. For example, the spanning tree and connected components codes in Lonestar use it in union-find operations.

The path-compression pattern is less frequent than the other five patterns but still occurs in several irregular algorithms. We included this pattern because it is the only common pattern we found that not only accesses direct graph neighbors but also the neighbors' neighbors, etc. *The resulting six patterns represent key low-level "dwarfs" of irregular graph codes* [31].

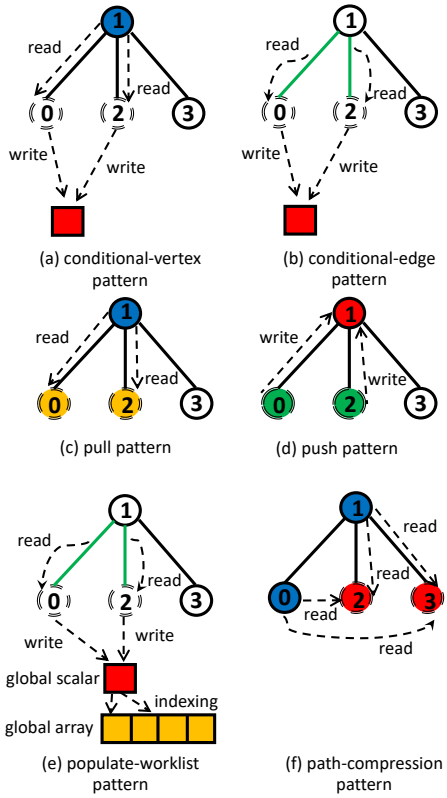

Fig. 3. Major irregular code patterns

Figure 3 visualizes each pattern. Squares represent shared global memory locations, circles denote graph vertices and vertex-local locations, solid lines are graph edges and edge-local locations, and dashed arrows track data flow. Red signifies shared write locations, blue demarcates shared read locations, yellow indicates non-shared write locations, and green marks non-shared read locations. The dashed circles outline two active vertices [32] that are processed in parallel.

The figure highlights potential sharing issues. The conditional edge pattern accesses a single shared read-modify-write location. The conditional vertex pattern does the same but also accesses multiple shared read-only locations. The pull pattern only accesses multiple shared read-only locations. The push pattern accesses multiple shared read-modify-write locations. The populate-worklist pattern accesses a single shared read-modify-write location as well as a single shared write-only array in which each element is written at most once. The path-compression pattern accesses multiple shared locations that are read and some of which are then written. In all cases that involve multiple shared locations, the memory accesses are indirect. Moreover, all six patterns include non-shared indirect accesses to the adjacency lists.

C. Pattern Variations

From each major pattern, we methodically create variations along several dimensions (where applicable). The first dimension is the data type of the shared memory locations. Indigo currently includes the following six types: signed 8-bit integers, unsigned 16-bit integers, signed 32-bit integers, unsigned 64-bit integers, 32-bit floats, and 64-bit doubles.

The second dimension is the neighbors being accessed. Indigo can process the adjacency lists in the following six ways: only the first neighbor, only the last neighbor, all neighbors in the forward direction, all neighbors in the reverse direction, the first few neighbors until a condition is met, and the last few neighbors until a condition is met. Although it does occur, accessing only the first or last neighbor is not very common. We still included these versions as they represent important corner cases (for example for bounds checks).

The third dimension is making the updates of the shared memory locations conditional. This increases the complexity, e.g., when trying to detect data races or out-of-bounds array accesses, because it introduces (additional) data-dependent control flow and makes the memory accesses more irregular.

The fourth dimension is inserting common bugs. We focus on two types: out-of-bounds memory accesses and synchronization errors. The out-of-bounds bugs involve going over the end of either of the two CSR arrays. The synchronization bugs involve making operations non-atomic that must be atomic, inserting performance-enhancing guards that introduce data races, and removing necessary barriers.

The fifth and final dimension is employing different parallel schedules. On the OpenMP side, this involves using a static or dynamic assignment of work to the threads. On the CUDA side, it involves assigning one vertex or multiple vertices to each processing entity (i.e., using persistent threads [33]), where a processing entity is a thread, a warp, or a block.

The five dimensions are orthogonal and can be combined in any way. Moreover, the bugs (in the fourth dimension) are independent of each other and any combination thereof can be present in the same code. Together, these combinations result in the thousands of microbenchmarks in Indigo.

D. Annotation Tags

Implementing a benchmark suite containing thousands of codes by hand is nearly impossible and not maintainable. Instead, we wrote just six source files per major pattern and express all variations in form of annotation tags. These tags are similar to the annotation comments in the Java Modeling Language (JML) [34]. Indigo automatically generates the OpenMP and CUDA codes from these annotated source files.

Listing 1 provides an excerpt of an annotated CUDA kernel. We use the syntax "/*@tag@*/" without the quotes to separate alternative statements on a line of code. Each annotated line can either be the code before the first tag, between the first and second tag, etc., or after the last tag. Tags with different names on different lines are *independent* and all combinations can be generated. For example, the alternatives before and after the 'reverse' tag will be combined with the alternatives before (empty) and after the 'break' tag, resulting in four versions. However, tags on different lines with the same name are *dependent*, meaning the same alternative will be used on all lines with the same tag names. For example, lines 2, 3, and 13 will all use their first, middle, or last alternative, thus only resulting in three versions. Together, the tags in Listing 1 express a total of 12 versions of this kernel. Listing 2 shows the version that is generated when the 'persistent' tag is enabled and all other tags are disabled. Note that the tag names are arbitrary strings that can be compared for equality.

```
1  int idx = threadIdx.x + blockIdx.x * blockDim.x;
2  int i = idx; /*@persistent@*/ /*@boundsBug@*/ int
       i = idx;
3  if (i < numv) { /*@persistent@*/ for (int i = idx;
        i < numv; i += gridDim.x * blockDim.x) {
     /*@boundsBug@*/
4    int beg = nindex[i];
5    int end = nindex[i + 1];
6    for (int j = beg; j < end; j++) { /*@reverse@*/
       for (int j = end - 1; j >= beg; j--) {
7      int nei = nlist[j];
8      if (i < nei) {
9        atomicAdd(data1, (data_t)1); /*@atomicBug@*/
         data1[0]++;
10       /*@break@*/ break;
11     }
12   }
13 } /*@persistent@*/ } /*@boundsBug@*/
```

Listing 1. Excerpt of Indigo source file for generating 12 versions of the conditional-edge pattern

```
1  int idx = threadIdx.x + blockIdx.x * blockDim.x;
2  for (int i = idx; i < numv; i += gridDim.x *
       blockDim.x) {
3    int beg = nindex[i];
4    int end = nindex[i + 1];
5    for (int j = beg; j < end; j++) {
6      int nei = nlist[j];
7      if (i < nei) {
8        atomicAdd(data1, (data_t)1);
9      }
10   }
11 }
```

Listing 2. One resulting CUDA version of the conditional-edge pattern

```
1  int beg = nindex[i];
2  int end = nindex[i + 1];
3  data_t val = 0;
4  for (int j = beg + threadIdx.x; j < end; j +=
       blockDim.x) {
5    val = max(val, data2[nlist[j]]);
6  }
7  val = __reduce_max_sync(~0, val);
8  if (lane == 0) s_carry[warp] = val;
9  __syncthreads(); /*@syncBug@*/
10 if (warp == 0) {
11   val = s_carry[lane];
12   val = __reduce_max_sync(~0, val);
13   if (lane == 0) {
14     /*@guardBug@*/ if (data1[0] < val) {
15       atomicMax(data1, val); /*@atomicBug@*/ data1
       [0] = max(data1[0], val);
16     /*@guardBug@*/ }
17   }
18 }
```

Listing 3. Excerpt of Indigo source file illustrating bug insertion

We use the tags to enable pattern variations, including inserting bugs. There are five different types of bugs. They are 'atomicBug', 'boundsBug', 'guardBug', 'raceBug', and 'syncBug'. We introduce them by removing necessary synchronization or allowing access past the end of an array. For example, the 'boundsBug' on line 3 of Listing 1 enables out-of-bound memory accesses by allowing the index i to exceed the array size on lines 4 and 5. The 'syncBug' on line 9 of Listing 3 removes a needed block-level barrier, the 'guardBug' on line 14 introduces a data race, and the 'atomicBug' on line 15 makes an update to a globally shared location non-atomic.

We believe it is important for the generated codes to be human readable. Thus, Indigo does not use synthetic variable names. It also automatically indents the code, which is necessary when variations introduce or remove *if* statements, and eliminates blank lines due to empty tags. The file name of each microbenchmark is the pattern name followed by all enabled tags to make it easy to identify which file contains which code.

E. Subset Selection

The large number of microbenchmarks and graphs in Indigo yields over a million possible combinations, which may take too long to run. Therefore, the suite provides the flexibility to generate user-defined subsets of the programs and inputs. This is done through two levels of configuration files. We chose this approach to simplify the subset selection for most users.

The first level is a *master list* of allowable parameter settings for each graph generator, including the range of graph sizes. It is meant for experienced users who can add and remove any valid parameter settings they like. Since editing this list requires knowledge about the parameters each graph generator takes, we opted to hide it from novice users.

The second level is a much simpler *configuration file* that we think anyone can easily understand and modify. It filters out unwanted code versions and input types and sizes. For example, the user can select to only generate bug-free codes and directed graphs with between 10 and 12 vertices. TACO [35] similarly creates tensor algebra kernels based on user-defined constraints. In this way, an Indigo user can generate a small subset for quick testing and later a more extensive subset to perform a detailed study.

The configuration file comprises one section to manage the code generation and another section to manage the graph generation as shown in Listing 4. Both sections consist of a number of rules, each specifying a set of selections.

```
1  CODE:
2     bug:          {hasbug}
3     pattern:      {pull, populate-worklist}
4     option:       {only_atomicBug}
5     dataType:     {int, float}
6
7  INPUTS:
8     direction:        {all}
9     pattern:          {~star}
10    rangeNumV:        {0-100, 2000}
11    rangeNumE:        {0-5000}
12    samplingRate:     50%
```

Listing 4. Sample configuration file

For ease of use, Indigo's configuration file lists all possible choices for each rule in form of a comment. These choices are also shown in Tables II and III. The shorthand notation "all" means all possible choices will be generated. The symbol "$\sim$" inverts the meaning of the selection. For example, "$\sim$star" means all graph types except for star graphs. Prefixing a choice with "only_" as in "only_atomicBug" means no other bug type can be present. There are no specific choices for the last three rules pertaining to the input generation. Instead, the user needs to provide one or multiple values or ranges of values. The sampling rate further controls the number of graphs generated and must be a single value. For example, a 50% rate means half of the graphs that meet the other four rules in the input section will actually be generated. Since the code and graph generators are deterministic, they will always produce the same suite for a given configuration regardless of what machine the generators run on. Indigo includes several example configuration files for building various small and large subsets. Users can choose the default, one of four provided, or their own filter to generate a subset.

TABLE II
CHOICES FOR MANAGING THE CODE GENERATION

Rule	**Choices**
Bug	all, hasbug, nobug
Pattern	all, conditional-vertex, conditional-edge, pull, push, populate-worklist, path-compression
Option	all, atomicBug, boundsBug, guardBug, raceBug, syncBug, break, cond, dynamic, last, persistent, reverse, traverse
Data type	all, int, char, double, float, long, short

TABLE III
CHOICES FOR MANAGING THE GRAPH GENERATION

Rule	**Choices**
Direction	all, directed, undirected
Pattern	all, DAG, k_max_degree, power_law, uniform_degree, all_possible_graphs, binary_forest, binary_tree, k_dim_grid, k_dim_torus, rand_neighbor, simple_planar, star
Sampling rate	value between 0% and 100%

V. EXPERIMENTAL METHODOLOGY

We used version 0.9 of Indigo to evaluate the verification tools listed in Table IV. ThreadSanitizer [36] is a dynamic data-race detector for C/C++ programs and is part of Clang 3.2 and gcc 4.8. Archer [37] is a data-race detector for OpenMP codes that combines static and dynamic techniques.

CIVL is a verification platform for parallel C programs. Its intermediate language, CIVL-C, employs a general model of concurrency that can represent OpenMP, CUDA, MPI, and Pthreads programs. CIVL includes front-ends to translate code to CIVL-C and a back-end that uses symbolic execution and model-checking techniques to verify CIVL-C programs.

Cuda-memcheck is a correctness checking suite for CUDA. It includes the memory access error and leak detection tool Memcheck [38], the shared memory data access hazard detection tool Racecheck [39], the unitialized global memory access

TABLE IV
TESTED VERIFICATION TOOLS

Tool	Version	OpenMP	CUDA
ThreadSanitizer [36]	9.3.1	Yes	No
Archer [37]	2.0.0	Yes	No
CIVL [42]	1.20	Yes	Yes
Cuda-memcheck [43]	11.4.0	No	Yes

detection tool Initcheck [40], and the thread synchronization hazard detection tool Synccheck [41].

The system we used for running the OpenMP codes has dual 10-core 3.1 GHz Xeon E5-2687W v3 CPUs. The CUDA codes were executed on a GeForce GTX Titan X GPU with 3072 processing elements in 24 multiprocessors. We ran the OpenMP experiments with 2 and 20 threads. For the CUDA experiments, we launch 2 blocks with 256 threads per block.

The operating system is Fedora 30, and the GPU driver version is 450.66. We used gcc 9.3.1 with the "-O3 -march=native -fopenmp" switches to compile the OpenMP codes and nvcc 11.0 with the "-O3" switch to compile the CUDA codes.

To keep the running times manageable, we excluded all data types other than 32-bit signed integers. This yielded 692 microbenchmarks. 254 are OpenMP and 438 are CUDA codes, including 146 OpenMP and 274 CUDA codes with bugs. We ran each of them with 209 generated graphs. These inputs comprise all possible undirected graphs ranging from 1 to 4 vertices and all other types of supported graphs with 29 and 773 (729 for the grids and tori) vertices. In total, we executed 106,172 tests for ThreadSanitizer and Archer as well as 91,542 tests for each Cuda-memcheck tool. Being a static tool, CIVL only verifies each code once. Since it can take a long time to analyze a microbenchmark, we only specified 2 threads for the CIVL OpenMP experiments.

As out-of-bound accesses may result in an infinite loop with the Racecheck tool, we do not use it on codes with this type of bug. Excluding it does not affect the results because none of the Indigo codes with this bug use the GPU's shared memory.

To evaluate each tool, we measured the four counts shown in Table V to produce a confusion matrix. A tool generates a false positive (FP) if it reports a non-existing bug. If it correctly detects an existing bug, it is a true positive (TP). It is a true negative (TN) if the tool does not detect any bug in a bug-free program. If it fails to detect an existing bug, it is a false negative (FN). Note that, for a bug-free program, a tool can only generate either an FP or TN result. Similarly, it can only generate either a TP or FN result for a buggy program.

TABLE V
CONFUSION MATRIX

	Bug-free code	Buggy code
Positive report	False positive (FP)	True positive (TP)
Negative report	True negative (TN)	False negative (FN)

To make the results easier to understand, it is common to convert them into the three higher-is-better metrics $accuracy$ (A), $precision$ (P), and $recall$ (R), which are defined as follows: $A = (TP + TN)/(TP + FP + TN + FN)$, $P = TP/(TP + FP)$, $R = TP/(TP + FN)$. The accuracy reflects the probability that the tool produces a correct report, the precision denotes the probability of correctly detecting a bug out of all positive reports, and the recall measures the probability of detecting a bug within all buggy codes.

VI. RESULTS

Table VI lists the raw counts we obtained for each evaluated tool. Table VII shows the corresponding accuracy, precision, and recall. The numbers in parentheses reflect the thread count.

TABLE VI
ABSOLUTE POSITIVE AND NEGATIVE COUNTS FOR EACH TOOL

Tool	Bug-free codes		Buggy codes	
	FP	TN	TP	FN
ThreadSanitizer (2)	5,317	17,255	14,829	15,685
ThreadSanitizer (20)	6,565	16,007	18,103	12,411
Archer (2)	2,587	19,985	8,471	22,043
Archer (20)	21,744	828	29,689	825
CIVL (OpenMP)	0	108	18	128
CIVL (CUDA)	0	164	64	210
Cuda-memcheck	0	34,276	17,406	39,860

TABLE VII
RELATIVE METRICS FOR EACH TOOL

Tool	Accuracy	Precision	Recall
ThreadSanitizer (2)	60.4%	73.6%	48.6%
ThreadSanitizer (20)	64.2%	73.4%	59.3%
Archer (2)	53.6%	76.7%	27.8%
Archer (20)	57.4%	57.7%	97.2%
CIVL (OpenMP)	49.6%	100.0%	12.1%
CIVL (CUDA)	52.1%	100.0%	23.4%
Cuda-memcheck	56.4%	100.0%	30.4%

The ThreadSanitizer and Archer results depend on the number of threads. They both have better accuracy and especially recall but lower precision with more threads. Since they are dynamic tools, they benefit from larger thread counts, which increase the chances of a bug manifesting itself, yielding a higher number of true positives. However, a larger number of threads also increases the observed interleavings and thus the analysis complexity, resulting in more false positives. ThreadSanitizer mostly outperforms Archer because we included an option in ThreadSanitizer to suppress bug detection outside of the parallel target kernel. Archer does not have such an option.

CIVL does not report any false positives, resulting in perfect precision. However, its accuracy and especially its recall are lower than those of Archer and ThreadSanitizer. This is due to CIVL still being under active development. It does not yet support several features that appear in our microbenchmarks, including "atomic capture" and "reduction" pragmas in OpenMP as well as atomic, warp-vote, and warp-shuffle functions in CUDA. Moreover, every microbenchmark with a missing atomic operation results in an internal CIVL error for the OpenMP codes[2]. For now, we count codes that use unsupported operations as negative results.

Cuda-memcheck also does not produce any false positives, yielding a perfect precision. Its accuracy and recall are better than CIVL's but mostly worse than ThreadSanitizer's and Archer's. Note, however, that we are comparing results from CUDA and OpenMP codes, only some of which are equivalent.

[2]We reported this bug (and the missing features) to the authors of CIVL.

A. Data-race Detection

Since ThreadSanitizer and Archer were designed for detecting data races, we provide results for just race detection in Table VIII. A false positive means the tool reports a data race but the program is race-free, though it may contain other types of bugs. Table IX shows the corresponding metrics.

TABLE VIII
RESULTS FOR DETECTING JUST OPENMP DATA RACES

Tool	No data races		Has data races	
	FP	TN	TP	FN
ThreadSanitizer (2)	6,764	23,332	12,196	10,794
ThreadSanitizer (20)	9,408	20,688	14,995	7,995
Archer (2)	3,497	26,599	6,009	16,981
Archer (20)	27,338	2,758	21,819	1,171

TABLE IX
METRICS FOR DETECTING JUST OPENMP DATA RACES

Tool	Accuracy	Precision	Recall
ThreadSanitizer (2)	66.9%	64.3%	53.0%
ThreadSanitizer (20)	67.2%	61.4%	65.2%
Archer (2)	61.4%	63.2%	26.1%
Archer (20)	46.3%	44.3%	94.8%

When detecting data races in the regular codes of the DataRaceBench suite [16], the accuracy, precision, and recall are 54.2%, 55.1%, and 95% for ThreadSanitizer and 83.3%, 91.2%, and 77.5% for Archer. Hence, Archer performs better on almost all metrics on regular codes. ThreadSanitizer has a lower accuracy and precision on the regular codes, which may be because we used the aforementioned suppression flag, but a higher recall. Overall, we find irregular codes to be at least as challenging as regular codes when detecting data races.

Interestingly, the results vary substantially between the six main code patterns. Table X shows the metrics of ThreadSanitizer with 20 threads split by pattern. There are no variations of the pull pattern in Indigo that contain data races. Evidently, the path-compression and the conditional-edge pattern make it easy to detect data races. In contrast, the conditional-vertex and especially the push pattern make it much harder. This highlights the importance of not only the multiple patterns but also including the same bug in each of them, that is, systematically creating variations of the irregular code patterns.

TABLE X
THREADSANITIZER METRICS FOR DETECTING JUST OPENMP DATA RACES IN DIFFERENT CODE PATTERNS

Pattern	Accuracy	Precision	Recall
Conditional-vertex pattern	49.9%	49.9%	70.8%
Conditional-edge pattern	88.4%	99.8%	76.9%
Push pattern	43.3%	44.7%	56.1%
Populate-worklist pattern	69.6%	99.1%	39.5%
Path-compression pattern	96.5%	100.0%	89.5%

Cuda-memcheck can only detect data races in the GPU's shared memory but not in global memory. Hence, we only show results for detecting races in shared memory. Table XI lists the counts. Table XII shows the corresponding metrics.

The Racecheck tool in Cuda-memcheck does not yield any false positives when detecting data races in shared memory. Its accuracy and precision are very high. Moreover, its accuracy and recall are roughly twice their counterparts in Table VII, indicating that the Racecheck tool performs quite a bit better on our codes than some of the other tools in Cuda-memcheck.

TABLE XI
CUDA-MEMCHECK COUNTS FOR DETECTING JUST CUDA DATA RACES IN SHARED MEMORY

Tool	No data races		Has data races	
	FP	TN	TP	FN
Cuda-memcheck	0	86,976	3,304	5,016

TABLE XII
CUDA-MEMCHECK METRICS FOR DETECTING JUST CUDA DATA RACES IN SHARED MEMORY

Tool	Accuracy	Precision	Recall
Cuda-memcheck	98.1%	100%	65.8%

B. Memory-error Detection

CIVL and Cuda-memcheck support detecting memory access errors. Tables XIII and XIV list the corresponding counts and metrics. Neither tool produces any false negatives. Note that an out-of-bound access only happens for some of the input graphs. Yet, both tools perform quite well on our CUDA codes.

TABLE XIII
COUNTS FOR DETECTING JUST MEMORY ACCESS ERRORS

Tool	No boundsBug		Has boundsBug	
	FP	TN	TP	FN
CIVL (OpenMP)	0	190	16	48
CIVL (CUDA)	0	326	64	48
Cuda-memcheck	0	68,134	14,102	9,306

TABLE XIV
METRICS FOR DETECTING JUST MEMORY ACCESS ERRORS

Tool	Accuracy	Precision	Recall
CIVL (OpenMP)	81.1%	100%	25.0%
CIVL (CUDA)	89.0%	100%	57.1%
Cuda-memcheck	89.8%	100%	60.2%

Again, the results vary quite a bit between the code patterns. Table XV shows the metrics for CIVL with 2 OpenMP threads. We did not evaluate any path-compression codes with out-of-bounds memory accesses. In the pull pattern, CIVL detects all memory errors perfectly. However, in the conditional-vertex, push, and populate-worklist patterns, it detects none of them. This again illustrates the need for including the same bug in different irregular code patterns when testing verification tools.

VII. SUMMARY AND CONCLUSIONS

Irregular programs tend to be data dependent, meaning that different inputs can result in very different runtime behavior. This paper presents Indigo, the first benchmark suite designed to enable extensive studies of such dynamic behavior in OpenMP and CUDA codes and to systematically exercise tools like program verifiers, compilers, and architectural simulators. Indigo is available at https://cs.txstate.edu/~burtscher/research/IndigoSuite/.

To create the suite, we extracted the most important *dwarf-like* code patterns from parallel graph analytics applications. We implemented Indigo to methodically generate hundreds

TABLE XV
CIVL METRICS FOR DETECTING JUST OPENMP OUT-OF-BOUND ERRORS IN DIFFERENT CODE PATTERNS

Pattern	Accuracy	Precision	Recall
Conditional-vertex pattern	75%	100%	0%
Conditional-edge pattern	87.5%	100%	50%
Pull pattern	100%	100%	100%
Push pattern	75%	100%	0%
Populate-worklist pattern	66.6%	100%	0%

of variations of each pattern, including some with planted bugs. We call the resulting codes "microbenchmarks". Indigo comes with generators that can produce an unbounded number of inputs for each microbenchmark, including all possible graphs with k vertices for systematic and exhaustive testing. Combining the thousands of codes with just as many inputs yields millions of distinct combinations to elicit a vast number of program behaviors. To control this number, Indigo allows the user to generate subsets of the code variations and inputs via their own or one of the provided configuration files.

We employed such a subset of over 100,000 experiments to evaluate several parallel-program verification tools. Our results show that bug detection tends to be more difficult in irregular codes than in regular codes. For example, ThreadSanitizer and Archer can detect 95% and 77.5% of the data races in the 'race-yes' regular programs from the DataRaceBench suite. However, on our short irregular codes, they only correctly detect 65.2% and 26.1% of the data races and produce false positives on many race-free programs. Moreover, we found the quality of these tools to vary greatly between different irregularity patterns. This highlights the need for including a variety of code patterns in irregular benchmark suites as well as the importance of including the same bug in different codes, that is, the importance of systematically creating variations of code patterns. We hope that our work will inspire others to build similar benchmark suites for additional domains.

ACKNOWLEDGMENTS

We thank the anonymous reviewers, Ganesh Gopalakrishnan, Stephen Siegel, Tanmay Tirpankar, and Alexander Wilton for their help and feedback to improve this paper.

REFERENCES

[1] S. A. Myers, A. Sharma, P. Gupta, and J. Lin, "Information network or social network? the structure of the twitter follow graph," in *Proceedings of the 23rd International Conference on World Wide Web*, 2014, pp. 493–498.

[2] D. J. Cook and L. B. Holder, "Graph-based data mining," *IEEE Intelligent Systems and Their Applications*, vol. 15, no. 2, pp. 32–41, 2000.

[3] C. Zhang, D. Song, C. Huang, A. Swami, and N. V. Chawla, "Heterogeneous graph neural network," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 793–803.

[4] S. Cyphers, A. K. Bansal, A. Bhiwandiwalla, J. Bobba, M. Brookhart, A. Chakraborty, W. Constable, C. Convey, L. Cook, O. Kanawi *et al.*, "Intel ngraph: An intermediate representation, compiler, and executor for deep learning," *arXiv preprint arXiv:1801.08058*, 2018.

[5] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in *2012 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2012, pp. 141–151.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 72–81.

[7] M. Kulkarni, M. Burtscher, C. Casçaval, and K. Pingali, "Lonestar: A suite of parallel irregular programs," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 2009, pp. 65–76.

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 2009, pp. 44–54.

[9] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (shoc) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, 2010, pp. 63–74.

[10] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, vol. 127, 2012.

[11] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *2012 innovative parallel computing (InPar)*. Ieee, 2012, pp. 1–10.

[12] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular gpgpu graph applications," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2013, pp. 185–195.

[13] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[14] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, "Graphbig: understanding graph computing in the context of industrial solutions," in *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.

[15] J. Gómez-Luna, I. El Hajj, L.-W. Chang, V. García-Floreszx, S. G. De Gonzalo, T. B. Jablin, A. J. Pena, and W.-m. Hwu, "Chai: Collaborative heterogeneous applications for integrated-architectures," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2017, pp. 43–54.

[16] C. Liao, P.-H. Lin, J. Asplund, M. Schordan, and I. Karlin, "Dataracebench: a benchmark suite for systematic evaluation of data race detection tools," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–14.

[17] Z. Xu, X. Chen, J. Shen, Y. Zhang, C. Chen, and C. Yang, "Gardenia: A graph processing benchmark suite for next-generation accelerators," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 15, no. 1, pp. 1–13, 2019.

[18] L. Dhulipala, J. Shi, T. Tseng, G. E. Blelloch, and J. Shun, "The graph based benchmark suite (gbbs)," in *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, 2020, pp. 1–8.

[19] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: an approach to modeling networks." *Journal of Machine Learning Research*, vol. 11, no. 2, 2010.

[20] L. Wang, W. Jie, and J. Chen, *Grid computing: infrastructure, service, and applications*. CRC Press, 2018.

[21] S. R. Blackburn and S. Gerke, "Connectivity of the uniform random intersection graph," *Discrete Mathematics*, vol. 309, no. 16, pp. 5130–5140, 2009.

[22] J. Dongarra, "Compressed row storage," http://www.netlib.org/utk/people/JackDongarra/etemplates/node373.html, accessed: 2021-7-3.

[23] G. Verma, Y. Shi, C. Liao, B. Chapman, and Y. Yan, "Enhancing dataracebench for evaluating data race detection tools," in *2020 IEEE/ACM 4th International Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE, 2020, pp. 20–30.

[24] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan, "Gklee: Concolic verification and test generation for gpus," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 215–224. [Online]. Available: https://doi.org/10.1145/2145816.2145844

[25] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson, "Gpuverify: a verifier for gpu kernels," in *Proceedings of the ACM international conference on Object oriented programming systems languages and*

applications, 2012, pp. 113–132.

[26] A. Eizenberg, Y. Peng, T. Pigli, W. Mansky, and J. Devietti, "Barracuda: Binary-level analysis of runtime races in cuda programs," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 126–140.

[27] T. Tu, X. Liu, L. Song, and Y. Zhang, "Understanding real-world concurrency bugs in go," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 865–878.

[28] T. Yuan, G. Li, J. Lu, C. Liu, L. Li, and J. Xue, "Gobench: A benchmark suite of real-world go concurrency bugs," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 187–199.

[29] S. Unkule, C. Shaltz, and A. Qasem, "Automatic restructuring of gpu kernels for exploiting inter-thread data locality," in *International Conference on Compiler Construction*. Springer, 2012, pp. 21–40.

[30] A. Qasem, A. M. Aji, and G. Rodgers, "Characterizing data organization effects on heterogeneous memory architectures," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2017, pp. 160–170.

[31] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, no. 10, p. 56–67, oct 2009. [Online]. Available: https://doi.org/10.1145/1562764.1562783

[32] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo *et al.*, "The tao of parallelism in algorithms," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 12–25.

[33] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style gpu programming for gpgpu workloads," in *2012 Innovative Parallel Computing (InPar)*, 2012, pp. 1–14.

[34] G. T. Leavens, "The java modeling language (jml)," *URL http://sourceforge. net/apps/wordpress/fixedptc*, 2007.

[35] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017. [Online]. Available: https://doi.org/10.1145/3133901

[36] "Threadsanitizer," https://github.com/google/sanitizers, accessed: 2021-6-28.

[37] "Archer," https://github.com/PRUNERS/archer, accessed: 2021-6-26.

[38] "Memcheck tool," https://docs.nvidia.com/cuda/cuda-memcheck/index.html/memcheck-tool, accessed: 2021-6-28.

[39] "Racecheck tool," https://docs.nvidia.com/cuda/cuda-memcheck/index.html/racecheck-tool, accessed: 2021-6-28.

[40] "Initcheck tool," https://docs.nvidia.com/cuda/cuda-memcheck/index.html/initcheck-tool, accessed: 2021-6-28.

[41] "Cuda-synccheck," https://docs.nvidia.com/cuda/cuda-memcheck/index.html/synccheck-tool, accessed: 2021-6-28.

[42] S. F. Siegel, M. Zheng, Z. Luo, T. K. Zirkel, A. V. Marianiello, J. G. Edenhofner, M. B. Dwyer, and M. S. Rogers, "Civl: the concurrency intermediate verification language," in *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.

[43] "Cuda-memcheck," https://docs.nvidia.com/cuda/cuda-memcheck/index.html, accessed: 2021-6-28.