Indigo3: A Parallel Graph Analytics Benchmark Suite for Exploring Implementation Styles and Common Bugs

YIQIAN LIU, Texas State University, USA
NOUSHIN AZAMI, Texas State University, USA
AVERY R VANAUSDAL, Texas State University, USA
MARTIN BURTSCHER, Texas State University, USA

Graph analytics codes are widely used and tend to exhibit input-dependent behavior, making them particularly interesting for software verification and validation. This paper presents Indigo3, a labeled benchmark suite based on 7 graph algorithms that are implemented in different styles, including versions with deliberately planted bugs. We systematically combine 13 sets of implementation styles and 15 common bug types to create the 41,790 CUDA, OpenMP, and parallel C programs in the suite. Each code is labeled with the styles and bugs it incorporates. We used 4 subsets of Indigo3 to test 5 program-verification tools. Our results show that the tools perform quite differently across the bug types and implementation styles, have distinct strengths and weaknesses, and generally struggle with graph codes. We discuss the styles and bugs that tend to be the most challenging as well as the programming patterns that yield false positives.

CCS Concepts: • **Software and its engineering → Software verification and validation**; • **Computing methodologies → Parallel computing methodologies**; **Parallel algorithms**.

Additional Key Words and Phrases: Benchmark-suite design, bug insertion, software verification, graph analytics, parallel computing

ACM Reference Format:
Yiqian Liu, Noushin Azami, Avery R VanAusdal, and Martin Burtscher. 2024. Indigo3: A Parallel Graph Analytics Benchmark Suite for Exploring Implementation Styles and Common Bugs. *ACM Trans. Parallel Comput.* 1, 1 (May 2024), 30 pages. https://doi.org/10.1145/3665251

1 INTRODUCTION

With the rise of social networks, recommender systems, GPS navigators, and data science, graph algorithms for computing communities, centrality, shortest paths, frequent motifs, and so on have become an important workload. Many of these algorithms exhibit irregular behavior, meaning their control flow and memory-access patterns are data dependent and tend to change during program execution [22]. Control-flow irregularity typically stems from *variable-iteration* loops, and memory-access irregularity usually comes from *pointer-chasing* operations.

Such behavior makes it challenging for verification tools to check program correctness, especially since 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 [21]. Parallelism often exacerbates the problem as the relative timing of the threads can change from run to run.

Authors' addresses: Yiqian Liu, y_l120@txstate.edu, Texas State University, 601 University Drive, San Marcos, TX, USA, 78666; Noushin Azami, Texas State University, 601 University Drive, San Marcos, USA, noushin.azami@txstate.edu; Avery R VanAusdal, Texas State University, 601 University Drive, San Marcos, USA, arv107@txstate.edu; Martin Burtscher, Texas State University, 601 University Drive, San Marcos, USA, burtscher@txstate.edu.

2329-4949/2024/5-ART $15.00
https://doi.org/10.1145/3665251

To make things worse, irregularity creates opportunities for implementing the same algorithm in many different ways. For example, we have written a connected-components (CC) algorithm using hundreds of different combinations of parallelization and implementation styles (168 CUDA versions, 36 OpenMP versions, and 36 C-threading versions) [45]. The large number of implementation styles adds yet another dimension of complexity to the program verification problem. In fact, the community possesses little understanding of how the many possible ways of implementing an irregular algorithm affect program verification.

Several widely-used benchmark suites with parallel implementations of irregular graph algorithms exist, including Lonestar [38] with 14 parallel implementations of 11 graph algorithms and Gardenia [69], an extended version of GAP [13], with 126 parallel implementations of 14 graph algorithms. These and similar suites include a range of interesting algorithms and inputs to study. However, none of them are designed to provide a large variety of each algorithm, nor do they include enough inputs to elicit the many different irregular behaviors needed to thoroughly evaluate the effectiveness of verification tools.

Moreover, since these suites were designed for performance measurements, they do not include bugs to help with designing and testing program verification tools. Only a few suites contain defective codes, such as DataRaceBench [43]. Hence, verification developers typically run their tools on existing open-source code bases [36]. This approach presents several challenges. First, it requires manual code inspection to verify any reported bugs. Second, it does not help with true or false negatives. Third, selecting a suitable set of open-source codes and installing them tends to be time consuming. Fourth, such codes naturally lack documentation of the bugs they contain. In some cases, tool designers have scanned commit histories to identify older versions of a code base with known bugs to test their tools [68]. However, this approach is even more time consuming, the "unfinished" code is even harder to install and run, and true and false negatives remain a problem. Clearly, the community could benefit from a "calibrated" suite that includes many code samples with *labeled* bugs to evaluate and improve their verification tools.

In response to this need, we introduced Indigo [47], a microbenchmark suite capable of automatically generating thousands of bug-free and buggy irregular parallel code patterns. While valuable, these microbenchmarks are simple in nature and do not compute meaningful results. To address this limitation, we expanded our efforts with the introduction of Indigo2 [45], which is based on 6 important graph algorithms and includes hundreds of bug-free CUDA, OpenMP, and parallel C++ implementations of each algorithm.

Building upon this foundation, we now present Indigo3, a fusion of the strengths of Indigo and Indigo2. Indigo3 extends Indigo2 by incorporating additional programs and versions, including a minimum spanning tree algorithm and hybrid parallelization of all codes, while also introducing a broad range of bugs akin to those found in Indigo. The incorporated software defects include data races, other synchronization issues, livelocks, deadlocks, and memory errors. Indigo3 methodically and automatically inserts these bugs as well as all possible combinations thereof to generate the codes in the suite. Since we manually select the applicable styles and bugs for each algorithm, all of the generated codes can be compiled. The bug-free codes generate identical results to the serial implementation of a validated algorithm. The file name of each code indicates which bugs, if any, are present. In total, Indigo3 includes 2516 bug-free codes and 39,274 buggy codes. In this paper, we use a subset of these codes to evaluate the effectiveness of current program verification tools and highlight important avenues for future work in the program verification domain.

The paper makes the following main contributions.

- It introduces Indigo3, the first *labeled* verification benchmark suite that includes a wide range of full-fledged buggy and bug-free irregular CUDA, OpenMP, and parallel C codes.

- It presents 13 largely orthogonal parallelization and implementation styles for CPUs and GPUs, yielding the 2516 bug-free versions of 7 key graph algorithms in Indigo3.
- It describes 15 types of common bugs and how they are systematically inserted into the bug-free base codes to create the 39,274 buggy programs in Indigo3.
- It evaluates 2 GPU and 3 CPU program verification tools on Indigo3 codes to explore how different implementation styles and bug types affect the tools' effectiveness.

The Indigo3 benchmark suite is publicly available in open source on Github [46].

The rest of the paper is organized as follows. Section 2 reviews relevant background information. Section 3 summarizes related work. Section 4 describes the design of the Indigo3 suite in detail. Section 5 discusses the experimental methodology. Section 6 evaluates several CPU and GPU program verification tools on buggy and bug-free codes from Indigo3. Section 7 summarizes the paper and draws conclusions.

2 BACKGROUND

This section provides background information on the main types of verification tools and the graph format used by the Indigo3 codes. It also presents an example of an irregular program.

2.1 Program verification

As outlined in the introduction, irregularity in programs is caused by input-dependent memory accesses and control flow. Such behavior makes codes harder to debug because even buggy codes will execute correctly for inputs that happen to yield (1) control flow that avoids the problematic code sections or (2) memory-access patterns that exclude the problematic data dependencies. In other words, only certain inputs may trigger the software defects present in the code. Moreover, the thread timing in parallel programs similarly only triggers software defects in some but not all executions of a program, even when using the same input. Together, this makes detecting bugs in irregular parallel programs particularly challenging.

Verification tools mainly consider the correctness of a program and are not concerned with performance. There are two main types of tools: static and dynamic. A dynamic tool observes runtime events while the program is executing [28]. Such tools tend to be relatively fast but only catch problems that actually occur during the observed run. For example, if the used input does not result in the code block containing a data race being executed, a dynamic tool will not detect the race. Hence, dynamic tools cannot prove the absence of data races even if they have not found any [43]. In other words, they typically produce no false positives but do produce false negatives.

Static verification tools, in contrast, examine the code before the program is run, for instance by analyzing the dependency graph, control flow, and data flow. Importantly, they consider all possible program behaviors and, in cases where they cannot prove that certain combinations of memory accesses or program paths never occur together, also include impossible behaviors. Hence, they typically produce no false negatives (if the bug lies in their search space) but do produce false positives. The generally large number and high complexity of code paths and memory-access patterns in irregular programs can quickly lead to a combinatorial explosion of possibilities to consider, making static tools potentially very slow on such codes.

In summary, irregular programs tend to be more challenging to verify than regular codes. This is true for both static and dynamic verification approaches.

2.2 Parallelization and implementation styles

There are numerous ways to parallelize irregular programs. We differentiate code optimizations from parallelization/implementation styles as follows. Parallelization and implementation styles are broadly applicable to many graph algorithms. In contrast, code optimizations tend to be specific to individual programs or a particular implementation of an algorithm. Due to this difference, programmers are more likely to be able to apply a given parallelization or implementation style when writing an irregular program than they are to apply a given code optimization. An example of a parallelization style is using thread, warp, or block granularity in GPU codes [73], as described in Section 4.1.8. An example of an implementation style is push versus pull (i.e., pushing data to neighboring vertices or pulling data from neighbors), which is common in both CPU and GPU graph codes [12], as described in Section 4.1.4.

Indigo3 employs numerous parallelization and implementation styles to create thousands of irregular programs. This multitude of combinations yields a wide range of irregular codes and behaviors for use in program verification and other domains. The styles present in Indigo3 are described in Section 4.1.

2.3 Irregular code example

Breadth-First Search (BFS) is an important graph traversal algorithm that is used in many applications, such as finding the shortest path in networks, identifying connected communities, and web crawling [51]. It labels all vertices with the shortest distance (in number of edges) from a given source vertex. Section 4 uses BFS as an example to describe different parallelization and implementation styles.

As shown in Algorithm 1, BFS starts by setting the distance of the source vertex to 0 and all other distances to ∞. For each $edge(v, n)$, a new distance is calculated (i.e., $dist[v] + 1$) in each iteration. Vertex n's distance is updated if the new distance is shorter. These edge relaxation operations repeat until the algorithm reaches a fixed point. The three *for all* loops are parallel assuming *dist* and *updated* are accessed with atomic loads and stores. Whereas more work-efficient BFS algorithms exist, this version generally yields more parallelism and is often used, especially in GPU codes.

Using the graph from Figure 1 as input and vertex 0 as the source, Table 1 shows the BFS computation step by step. It initializes the distance of the source to 0 and all other distances to ∞. In the first iteration, every active vertex v (i.e., whose distance is not ∞) calculates a new distance (i.e., $dist[v] + 1$) to its neighbors. The new distance for vertices 1 and 2 is 1, which is smaller than their current distances, so they are updated to 1, as shown in the *Iter1* column of the table. Similarly, in the second iteration, vertices 0, 1, and 2 calculate new distances to their neighbors and find shorter distances for vertices 3 and 4. The next iteration is the final iteration because no new shorter distances are found.

Table 1. Distance values computed in each step of the BFS algorithm on the example graph

Vertex	Init	Iter1	Iter2	Iter3
0	0	0	0	0
1	∞	1	1	1
2	∞	1	1	1
3	∞	∞	2	2
4	∞	∞	2	2

Note that this algorithm is input dependent and has both control-flow (e.g., line 12) and memory-access (e.g., line 14) irregularity. It is impossible to statically predict the iteration count of the inner *for-all* loop without knowing the input graph. Similarly, it is impossible to statically predict the order in which the elements of the *dist* array will be written unless we know the input graph and the order of the elements in the adjacency lists.

Algorithm 1 Parallel breadth-first search

```
Require: Graph G = (V, E) and source vertex s
 1: for all vertices v ∈ V do
 2:     if v = s then
 3:         dist[v] ← 0
 4:     else
 5:         dist[v] ← ∞
 6:     end if
 7: end for
 8: updated ← true
 9: while updated do
10:     updated ← false
11:     for all vertices v ∈ V do
12:         for all neighbors n ∈ adj[v] do
13:             if dist[n] > dist[v] + 1 then
14:                 dist[n] ← dist[v] + 1
15:                 updated ← true
16:             end if
17:         end for
18:     end for
19: end while
Ensure: Each vertex is labeled with the shortest distance from s
```

Implementing the loop over a vertex's neighbors (line 12) using the CSR format (see below) provides the opportunity for out-of-bounds accesses, especially in the presence of vertices with no neighbors. Moreover, the writes to the *dist* array as well as to *updated* are likely to yield data races in a parallel implementation unless proper synchronization primitives are utilized. For example, assume two threads are processing the graph from Figure 1. Since vertex 4 is a neighbor of vertices 2 and 3, a data race is possible if the two threads processing vertices 2 and 3, respectively, are allowed to push their updated distance to vertex 4 in an unsynchronized manner. Depending on internal timing, the distance of vertex 4 may end up as the distance from vertex 2, vertex 3, or some other value, even a seemingly impossible arbitrary value [18].

2.4 CSR graph format

The Compressed Sparse Row (CSR) format is one of the most widely used graph representations [27]. It is based on two dense arrays: an array of indices and an array of edges. The edge array holds the concatenated adjacency lists of all vertices. The index array holds the starting position (index) of each adjacency list. It has an extra element at the end specifying the size of the edge array. Figure 1 shows an example graph and its CSR representation.

For example, Pannotia [23] and Lonestar [38] use CSR inputs. All Indigo2 and, by extension, Indigo3 input graph generators produce graphs in this format, meaning that every generated graph can be used as an input for any code in our suites. Moreover, basing Indigo3 on the CSR format makes it easy for users to use their own graphs. For this purpose, we provide converters from several common formats (e.g., MatrixMarket, SNAP, and DIMACS) to our CSR format [20].

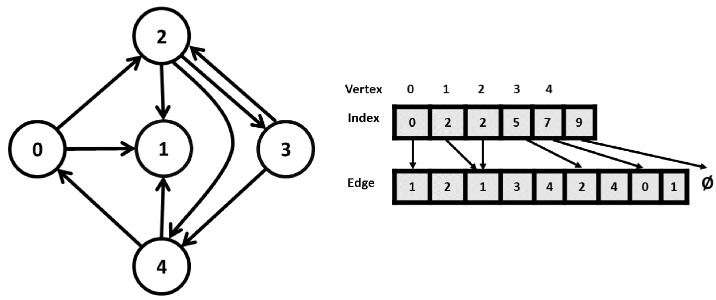


Fig. 1. Example graph (left) and corresponding CSR representation (right)

3 RELATED WORK

This section reviews prior benchmark suites of parallel programs (designed for either performance evaluation or verification), automatic code generation, and verification tools for parallel codes.

3.1 Parallel benchmark suites

Many benchmark suites with parallel codes exist. They target a plethora of program behaviors, application domains, programming languages, and so on. The early suites that focus on parallel programs mainly comprise *regular* high-performance computing (HPC) applications. One of the first suites not focusing on HPC is PARSEC [16], released in 2008, which contains 12 regular parallel codes. With accelerators becoming popular, quite a few suites now include GPU code. The Rodinia [24] suite targets heterogeneous systems. It exhibits different types of parallelization, memory-access and data-communication patterns, synchronization, and power consumption through 23 regular parallel codes written in CUDA, OpenMP, and OpenCL. The SHOC [25] suite is designed to test the performance and stability of heterogeneous systems. It contains 25 regular parallel codes. Parboil [61] 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. It includes 11 parallel codes. The Chai [32] suite includes 14 parallel codes to evaluate 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. Lonestar [38] contains 22 C++ and CUDA implementations of iterative graph algorithms. Pannotia [23] is an OpenCL suite of 8 applications for studying graph algorithms on GPUs. GraphBIG [50] 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 for 6 mostly irregular parallel codes written in OpenMP. GARDENIA [69] is a suite for studying irregular graph algorithms on accelerators. It includes 9 workloads from graph analytics, sparse linear algebra, and machine learning. GBBS [26] is a C++ suite of scalable, provably-efficient implementations of 20 graph problems for shared-memory multicore machines. It extends the Ligra interface with additional primitives and clearly defined cost bounds. Our Indigo3 suite, which is based on irregular graph algorithms, is much larger than these prior suites. It contains 2516 bug-free and 39,274 buggy codes.

There are also parallel benchmark suites in other domains. For instance, the NAS Parallel Benchmarks for GPUs (NPB-GPU) [10] contain larger CFD applications with more complex routines offloaded to the GPU. SPar [33] is a Domain-Specific Language (DSL) for developing parallel stream applications. It uses standard C++ attributes to introduce annotations for tagging components such as the stream sources and processing stages. Stream processing introduces a unique set of challenges, including ensuring the correct order (e.g., video applications need to keep the order of the frames). SPBench [30] is a framework for benchmarking such stream processing applications.

Many prior publications present ways to parallelize and optimize irregular graph codes. Several of them discuss and evaluate at least some implementation styles, but no systematic study of a large number of styles exists. Becchi et al. propose workload consolidation schemes [67] and different parallelization templates [41] to increase the GPU utilization of programs with nested parallelism. Wang et al. characterize dynamically formed parallelism and evaluate codes designed to exploit them [66]. Nasre et al. present morph algorithms and provide insights into how other morph algorithms can be efficiently implemented for GPUs [56]. In contrast, Indigo3 systematically applies 13 general parallelization and implementation styles to a set of 7 key graph algorithms.

Indigo3 not only includes orders of magnitude more codes than other benchmark suites but also a much larger number of inputs (which is important for data-dependent codes) and supports the creation of user-defined subsets through configurable code and graph generators. Between the thousands of codes and the unbounded number of inputs, Indigo3 allows users to run millions of distinct tests and to create subsets for many different usage scenarios. Furthermore, as described below, Indigo3 includes versions of its codes with deliberately planted bugs, giving users the ability to methodically test and analyze program verification tools.

3.2 Benchmark suites for data-race detection

DataRaceBench [43] 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 verifiers such as 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 [64]. RMARaceBench [59] is a microbenchmark suite to evaluate the capabilities of RMA (Remote Memory Access) race detection tools for MPI RMA, OpenSHMEM, and GASPI. It consists of about 100 synthetic race test cases for each programming model, aiming to cover all possible race scenarios. In our prior work [48], we introduced the Indigo benchmark suite, which contains common irregular code patterns. We systematically built variations of these patterns to alter the control-flow and memory-access behavior and/or to introduce bugs, yielding the thousands of OpenMP and CUDA microbenchmarks in the suite. In contrast, Indigo3 includes full-fledged graph algorithms instead of only short parallel code patterns. This enabled us to introduce additional parallelization bugs, yielding over 41,000 codes for verification-tool evaluation.

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 [62]. GoBench [71], 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. Indigo3's configuration file similarly defines the types of codes and inputs to be included in the generated suite.

3.3 Automatic code generation

The source code annotation and variation in CREST [63] and DLBENCH [58] inspired the code generation process in the Indigo suites. 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 our prior work on parallelization and implementation styles for graph algorithms, we took 6 key graph algorithms, generated hundreds of CUDA, OpenMP, and parallel C++ versions of each of them, and published them in the Indigo2 suite [45]. To determine which styles work well and under what circumstances, we evaluated 1106 of the Indigo2 programs on various systems and inputs. Most if not all of these styles have separately been described before. For example, Hong et al. [35] propose a warp-centric programming method to improve the performance of applications with heavily imbalanced workloads. Nasre et al. study data-driven and topology-driven implementations to understand the tradeoffs [54] and investigate high-level methods to eliminate atomics in irregular programs [52]. Pingali et al. discuss different styles to process nodes (e.g., topology-driven and data-driven) and operators that modify the graph (e.g., morphs and local computations) [57]. Indigo2 combines these styles in hundreds of different ways, most of which have never been studied before. Indigo3 goes a step further by introducing bugs into the codes of Indigo2 to enable the evaluation of verification tools. Moreover, we ported the C++ codes from Indigo2 to C code in Indigo3 because many program verification tools do not yet support C++.

3.4 Program verifiers

GKLEE [42] searches for correctness and performance bugs in GPU codes. It includes 40 benchmarks that cover many CUDA program behaviors and problems such as thread divergence, bank conflicts, deadlock, and data races. GPUVerify [15] comes with a suite of 163 CUDA and OpenCL kernels drawn from public and commercial resources. Barracuda [29] 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 third-party verification suites with buggy GPU codes exist, all of these tools include their own. ThreadSanitizer [8] is a dynamic data-race detector for C/C++ programs and is part of Clang 3.2 and gcc 4.8. Archer [1] is a data-race detector for OpenMP codes that combines static and dynamic techniques. CIVL [60] 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. Compute-sanitizer (formerly cuda-memcheck) is a correctness-checking suite for CUDA. It includes the memory access error and leak detection tool Memcheck [5], the shared memory data access hazard detection tool Racecheck [6], the unitialized global memory access detection tool Initcheck [4], and the thread synchronization hazard detection tool Synccheck [7]. We evaluate several of these CPU and GPU program verification tools in the result section.

4 INDIGO3 DESIGN

The following subsections describe the various parallelization and implementation styles included in the Indigo3 programs. We illustrate each style on the example of the breadth-first-search algorithm described in Section 2.3. Note that, throughout this paper, we assume the shared data values (e.g., the distances) to be scalars and assume load and store instructions to atomically read and write these values [19].

We wrote our graph codes using three parallel programming models: CUDA, OpenMP, and C threads. CUDA programs operate at multiple levels of parallelism. 32 contiguous threads form a warp and execute the same instruction in the same cycle (or are disabled). Sets of up to 32 warps (up to 1024 threads) form a block, and the blocks are grouped into a grid. CUDA provides built-in variables for the thread and block indices as well as the block and grid dimensions. These values are often combined by computing $threadIdx.x + blockIdx.x * blockDim.x$ to form a global index for assigning work to each thread, which we call *gidx* in our codes. OpenMP is based on *pragma* compiler directives. Each such directive consists of a name followed by optional clauses. For example, a clause can specify the scheduling to be used or a reduction operation. In Listing 11b below, it selects dynamic scheduling. Since C11, C supports multithreading in the standard library. It includes built-in types and functions for threads, atomics, mutual exclusion, and more.

4.1 Parallelization and implementation styles

This section describes the parallelization and implementation styles available in Indigo3.

4.1.1 *Vertex-based vs. edge-based.*

Graphs can be processed by iterating across either their vertices or their edges [72]. Listing 1a shows vertex-based code, where every thread processes a different vertex v based on the unique global thread index (*gidx*) and iterates over all neighbors n of v. Listing 1b shows edge-based code that assigns a different edge $e = (v, n)$ to each thread.

The algorithm to be implemented and the graph representation (e.g., CSR format [31]) typically determine which style is preferable. For instance, if the graph is represented by a set of adjacency lists, it is often more natural to employ the vertex-based style. To streamline the discussion, we use this style in the following subsections.

(a) Vertex-based

```
v = gidx;
if (v < nodes) {
  beg = nbr_idx[v];
  end = nbr_idx[v + 1];
  for (i = beg; i < end; i++) {
    n = nbr_list[i];
    ...
} }
```

(b) Edge-based

```
e = gidx;
if (e < edges) {
  v = src_list[e];
  n = dst_list[e];
  ...
}
```

Listing 1. Vertex- and edge-based computations

4.1.2 *Topology-driven vs. data-driven.*

This style describes two ways to determine which data-structure elements to process [57]. The topology-driven approach in Listing 2a simply processes all elements. In contrast, the data-driven approach in Listing 2b only processes the elements that likely need to be updated, which are stored in a worklist (*wl*). For example, topology-driven BFS applies the relaxation function to all vertices of the graph in each iteration. Data-driven BFS only applies the relaxation function to the vertices in the worklist. Those vertices are in the worklist because their distance changed in the prior iteration.

The topology-driven style tends to yield more parallelism and is easier to implement. The data-driven style is more work efficient and, therefore, often results in better performance, especially for iterative algorithms that operate on high-diameter graphs.

4.1.3 *Duplicates in worklist vs. no duplicates in worklist.*

This style, which only applies to data-driven implementations, specifies whether or not duplicate items are allowed on the worklist [55]. In codes that allow duplicates, as shown in Listing 3a, each thread can push a vertex onto the worklist regardless of whether the worklist already contains that vertex. In programs that do not allow duplicates, as shown in Listing 3b (where *itr* denotes the current iteration), the threads may only add a vertex to the worklist if it is not already there.

(a) Topology-driven

```
v = gidx;
if (v < nodes) {
  ...
}
```

(b) Data-driven

```
idx = gidx;
if (idx < worklist_size) {
  v = worklist[idx]
  ...
}
```

Listing 2. Topology- and data-driven computations

Disallowing duplicates eliminates redundant work in the next iteration. Moreover, it caps the size of the worklist. However, it incurs additional synchronization overhead and requires extra state tracking (*stat*) to determine whether a vertex is already on the worklist.

(a) Duplicates in worklist

```
idx = atomicAdd(&worklist_size, 1);
worklist[idx] = v;
```

(b) No duplicates in worklist

```
if (atomicMax(&stat[v], itr) != itr) {
  idx = atomicAdd(&worklist_size, 1);
  worklist[idx] = v;
}
```

Listing 3. Duplicates and no duplicates in worklist

4.1.4 Push vs. pull.

The data flow in programs that update vertex data can be either push-based, where data is pushed from a vertex to its neighbors, or pull-based, where data is pulled from the neighbors to the vertex [14]. For example, in push-style BFS, shown in Listing 4a, a thread reads the vertex distance, adds 1, and updates the neighbor if the new distance is shorter. In pull-style BFS, shown in Listing 4b, the thread reads the neighbor's distance, adds 1, and updates the vertex distance if it is shorter.

Using the push style, different threads may update the same neighboring vertex. In contrast, the pull style guarantees that there is only a single writer per vertex. Moreover, it allows the update to be factored out of the loop (not done in Listing 4b), thus reducing memory accesses. Having said that, push is sometimes a more natural fit for the underlying algorithm and preferred in combination with a data-driven approach because only the neighbors that were actually updated need to be placed on the worklist.

(a) Push

```
for (i = beg; i < end; i++) {
  n = nbr_list[i];
  new_dist = dist[v] + 1;
  atomicMin(&dist[n], new_dist);
}
```

(b) Pull

```
for (i = beg; i < end; i++) {
  n = nbr_list[i];
  new_dist = dist[n] + 1;
  atomicMin(&dist[v], new_dist);
}
```

Listing 4. Push and pull data flow

4.1.5 Read-write vs. read-modify-write.

Many graph algorithms conditionally update vertex data, where a thread reads the current value, performs a computation with it, and writes the new value if it meets a certain condition. For example, in BFS, the vertex distance is only updated if the new distance is shorter. This read-write approach works in certain situations, such as in Listing 5a, because the updates are monotonic and the algorithm is resilient to temporary priority inversions [53]. The read-modify-write style shown in Listing 5b is more general as it does not suffer from this problem, but it requires an atomic read-modify-write operation, which tends to be slower and hampers parallelism.

(a) Read-write

```
old_dist = dist[v];
if (new_dist < old_dist)
  dist[v] = new_dist;
```

(b) Read-modify-write

```
atomicMin(&dist[v], new_dist);
```

Listing 5. Read and write operations

4.1.6 Non-deterministic vs. deterministic.

The unpredictable timing of threads can introduce internal non-determinism in some parallel codes [17]. In Listing 6a, multiple threads may write an element of the *dist* array that is read by another thread. Depending on which thread performed the last write before the read, a different value may be read, leading to the computation of a different new distance. Any non-final distance value will be overwritten in subsequent iterations, meaning the ultimate result is deterministic, but the number of iterations may differ from run to run. Note that we only study programs in this paper where the final result is deterministic.

To make the code internally deterministic, Listing 6b uses two arrays, one that is only read ($dist1$) and another that is updated ($dist2$). However, in this approach, the computation can no longer take advantage of results generated in the same iteration, which may slow down the execution. On the upside, the deterministic code will always require the same number of iterations for a given input, which can simplify debugging [11].

(a) Non-deterministic

```
new_dist = dist[v] + edge_weight;
atomicMin(&dist[n], new_dist);
```

(b) Deterministic

```
new_dist = dist1[v] + edge_weight;
atomicMin(&dist2[n], new_dist);
```

Listing 6. Non-deterministic and deterministic updates

4.1.7 Persistent vs. non-persistent.

This style only applies to GPU codes. The persistent style, shown in Listing 7a, uses as many threads as the GPU can concurrently schedule on its SMs [34], meaning a thread may need to process multiple vertices (as is done in CPU codes). In contrast, the non-persistent style in Listing 7b launches at least as many threads as the input has vertices and assigns no more than one vertex to each thread. For graphs where the number of vertices exceeds the number of threads that can concurrently run on the SMs, the GPU will automatically schedule batches of threads until all threads have executed. The persistent style is a little more complex to implement but may improve performance in cases where common subexpressions can be precomputed or common data preloaded and then reused.

4.1.8 Thread vs. warp vs. block.

This variation only applies to GPU codes. It refers to the granularity at which the program processes the vertices. Threads, warps, and blocks are the three hardware-supported granularities. In thread-based BFS, each thread processes all neighbors of a vertex as shown in Listing 8a. In warp- or block-based BFS, the entire warp or block processes the neighbors of a single vertex, respectively, as

(a) Persistent

```
threads = blockDim.x * gridDim.x;
for (v = gidx; v < nodes; v += threads)
  ...
```

(b) Non-persistent

```
v = gidx;
if (v < nodes)
  ...
```

Listing 7. Persistent and non-persistent threads

shown in Listings 8b and 8c. Both warp- and block-based processing yields a two-level parallelization scheme: the vertices are distributed across the warps or blocks while the neighbors are distributed across the threads within the warp or block. This approach is useful for reducing load imbalance when processing high-degree vertices in power-law graphs [9]. However, it is typically not useful for low-degree graphs such as road networks.

(a) Thread

```
beg = nbr_idx[v];
end = nbr_idx[v + 1];
for (i = beg; i < end; i++)
  ...
```

(b) Warp

```
lane = threadIdx.x % warpSize;
beg = nbr_idx[v];
end = nbr_idx[v + 1];
for (i = beg + lane; i < end; i += warpSize)
  ...
```

(c) Block

```
beg = nbr_idx[v];
end = nbr_idx[v + 1];
for (i = beg + threadIdx.x; i < end; i += blockDim.x)
  ...
```

Listing 8. Thread, warp, and block parallelization

4.1.9 *Global-add vs. block-add vs. reduction-add.*

Reductions are widely used in parallel computing to combine multiple independently computed partial results into a single global result using a binary associative operator [44]. For example, multiple threads may need to add the partial sums they computed to a global sum.

We employ three reduction styles in our GPU codes. The first approach directly updates a shared global variable using atomic operations, as shown in Listing 9a. The second approach makes use of faster block-level atomics. All threads of a block first compute a block-local solution in the GPU's "shared memory", and only one thread updates the global solution as shown in Listing 9b. This minimizes the number of slower global atomics. The third approach utilizes not only shared-memory buffers for local results but also warp-level primitives to quickly perform warp and block reductions as outlined in Listing 9c. This implementation is more complex but tends to be faster as it avoids most memory accesses.

4.1.10 *Atomic-reduction vs. critical-reduction vs. clause-reduction.*

We also employ three reduction styles in our CPU codes. OpenMP and C provide atomic operations, enabling each thread to atomically update a shared variable, as shown in Listing 10a. Mutexes are also supported, allowing the programmer to update shared variables in critical sections, as shown in Listing 10b. Additionally, OpenMP provides a reduction clause, as shown in Listing 10c. Using a critical section typically results in substantial overhead and poor performance, but it is the most general of the three approaches. The reduction clause tends to produce the fastest code.

(a) Global-add

```
atomicAdd(&ctr, val);
```

(b) Block-add

```
atomicAdd_block(&block_ctr, val);
__syncthreads(); // block barrier
if (threadIdx.x == 0)
  atomicAdd(&ctr, block_ctr);
```

(c) Reduction-add

```
warp_ctr = warp_reduction(val);
__syncthreads(); // block barrier
block_ctr = block_reduction(warp_ctr);
__syncthreads(); // block barrier
if (threadIdx.x == 0)
  atomicAdd(&ctr, block_ctr);
```

Listing 9. Different reductions in CUDA

(a) Atomic reduction

```
#pragma omp parallel for
for (i = beg; i < end; i++) {
  ...
  #pragma omp atomic
  sum += val;
}
```

(b) Critical reduction

```
#pragma omp parallel for
for (i = beg; i < end; i++) {
  ...
  #pragma omp critical
  sum += val;
}
```

(c) Clause reduction

```
#pragma omp parallel for reduction(+: sum)
for (i = beg; i < end; i++) {
  ...
  sum += val;
}
```

Listing 10. Different reductions in OpenMP

4.1.11 *Default scheduling vs. dynamic scheduling.*

OpenMP provides a convenient way to parallelize certain *for* loops using a *parallel for* directive. By default, as shown in Listing 11a, this directive statically assigns each thread a contiguous chunk of loop iterations. In contrast, the dynamic schedule in Listing 11b assigns iterations at runtime whenever a thread is ready to execute another iteration. This improves the load balance but incurs overhead.

(a) Default scheduling

```
#pragma omp parallel for
for (v = 0; v < nodes; v++) {
  ...
}
```

(b) Dynamic scheduling

```
#pragma omp parallel for schedule(dynamic)
for (v = 0; v < nodes; v++) {
  ...
}
```

Listing 11. Default and dynamic loop scheduling

4.1.12 *Blocked vs. cyclic.*

When parallelizing the iterations of a *for* loop, a blocked schedule assigns a contiguous chunk of iterations to each thread, as shown in Listing 12a. If the iterations' running times correlate with their loop index, a block distribution can lead to load imbalance. The cyclic schedule in Listing 12b assigns the iterations in a round-robin fashion to the threads, which improves the load balance in this scenario. A blocked schedule usually has better data locality in CPUs because each thread accesses contiguous memory locations. However, a cyclic schedule yields better data locality in GPUs because of coalesced memory accesses, i.e., combining multiple memory accesses into a single memory transaction.

(a) Blocked scheduling

```
beg = tid * nodes / threads;
end = (tid + 1) * nodes / threads;
for (v = beg; v < end; v++) {
  ...
}
```

(b) Cyclic scheduling

```
for (v = tid; v < nodes; v += threads) {
  ...
}
```

Listing 12. Blocked and cyclic scheduling

4.2 Common bugs

As discussed in the background section, the input-dependent behavior makes bug detection particularly challenging in irregular codes. Additionally, certain parallelization bugs, such as data races, can be difficult to find because they are thread-timing dependent and may not manifest every time the code is executed. To help the community develop better tools and techniques to identify such bugs, Indigo3 contains versions of all its codes with intentionally planted software defects, including parallelism bugs (e.g., data races, missing barriers, livelock, and deadlock), memory bugs, and other serial bugs. Table 2 lists the parallelism-related bug types, Table 3 the memory bug types, and Table 4 the remaining bug types available in Indigo3.

Table 2. Parallelism bug types

Name	**Description**	**Bug-free example**	**Buggy example**
RaceBug	Missing atomic operation	atomicAdd(val, 1);	val++;
SyncBug	Missing barrier	syncthreads();	//no barrier
MixSyncBug	Mixing synchronization	critical(dist[src], s); critical(dist[dst], d);	critical(dist[src], s); atomic(dist[dst], d);
LivelockBug	Actively running w/o progress	if (newd < d) then d = newd;	if (newd <= d) then d = newd;

DeadlockBug	Some threads wait forever	if (v < nodes)
		then ...;
		syncthreads();
			if (v < nodes)
			then syncthreads();

GuardBug	Non-atomic check	atomicMax(d, m);	if (d < m) then atomicMax(d, m);

Most of these bug types are well known. The GuardBug is a data race where a variable is accessed both atomically and non-atomically (e.g., in an attempt to avoid the slower atomic operation when it is not needed). Unlike the BoundsBug, the NbrBoundsBug often does not result in accesses past the end of an array but only past the end of one of the concatenated adjacency lists in the CSR's edge array (see Figure 1), making it harder to detect. The WorkloadBug occurs when the problem size is not evenly divisible by the number of threads. It ends up not processing all of the workload.

Each bug is independent in the sense that it causes a software defect no matter if there are any other bugs in the code. However, one bug may interact with another and yield more complex program behavior. For example, the memory bug "BoundsBug" can lead to out-of-bounds accesses,

Table 3. Memory bug types

Name	Description	Bug-free example	Buggy example
NameBug	Wrong variable	for (...; v < nodes; ...)	for (...; v < edges; ...)
ExcessThreadsBug	Too many threads	if (gidx < nodes)	//no check
BoundsBug	Out-of-bounds access	type buffer[size]; a = buffer[size - 1];	type buffer[size]; a = buffer[size];
NbrBoundsBug	Exceeding adjacency list	for (...; nbr < end; ...)	for (...; nbr <= end; ...)
UninitializedBug	Data not fully initialized	data[v] = init;	//no initialization
ShadowBug	Re-declaring a variable in an inner scope	int i; for (i = v; ...);	int i; for (int i = v; ...);

Table 4. Other bug types

Name	Description	Bug-free example	Buggy example

OverflowBug	Range overflow	val = INT_MAX;
		if (val != INT_MAX)
		then val += d;
			val = INT_MAX;
			val += d;

WorkloadBug	Incorrect work assignment	gidx * size / threads;	chunksize = size / threads; gidx * chunksize;

which may trigger race conditions if multiple threads access the same out-of-bounds memory address. Hence, combining “BoundsBug” with “RaceBug” may increase the chance of data races.

Note that combining bugs increases the number of codes exponentially. For example, 3 bugs yield 7 buggy combinations (3 versions with 1 bug, 3 versions with 2 bugs, and 1 version with 3 bugs). Hence, adding just 3 bugs results in 7 times more codes than there are bug-free codes. Since at least 3 of the 14 bugs listed in Tables 2, 3, and 4 are applicable to each of our bug-free codes, we end up with nearly 40,000 buggy codes in Indigo3.

4.3 Annotation tags

Combining the implementation styles and bugs yields thousands of codes for each algorithm, making it nearly impossible and not maintainable to produce them by hand. Hence, we wrote just a few source files per algorithm and expressed all variations using annotation tags. These tags are similar to the annotation comments in the Java Modeling Language (JML) [40]. Indigo3 automatically generates the codes from the annotated source files. This code generation framework enables us and others to easily introduce additional implementation styles and bugs in the future by adding more tags.

Listing 13 provides an excerpt of annotated CUDA code. We use the syntax “/*@tag@*/” (without the quotes) to label alternative statements on a line of code. Each tag is associated with the code that follows it. The associated code will be generated when the tag is activated. Only one tag per line can be active at a time. Tags with different names on different lines are *independent* and all combinations can be generated. 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. Furthermore, matching tags affixed with “+” and “-”, such as Lines 3 and 5 in Listing 13, extend the activation idea to a block of code and enable the nesting of tags. This provides more flexibility and allows us to express complex interactions between tags. Listing 14 shows the generated codes for the persistent and non-persistent style that have no name bug and no bounds bug.

```
/*@NoNameBug@*/ const int gsize = nodes; /*@NameBug@*/ const int gsize = edges;

/*@+NonPersist@*/
/*@NoBoundsBug@*/ if (v < gsize) { /*@BoundsBug@*/ if (v <= gsize) {
/*@-NonPersist@*/

/*@+Persist@*/
/*@NoBoundsBug@*/ for (idx = v; idx < gsize; idx += threads) { /*@BoundsBug@*/ for
    (idx = v; idx <= gsize; idx += threads) {
/*@-Persist@*/
    ...
}
```

Listing 13. Tag-based annotations to generate code variations

(a) Non-persistent code example

```
const int gsize = nodes;
if (v < gsize) {
  ...
}
```

(b) Persistent code example

```
const int gsize = nodes;
for (idx = v; idx < gsize; idx += threads) {
  ...
}
```

Listing 14. Examples of generated code

We believe it is important for the generated codes to be human readable so they can be manually inspected and understood. Thus, Indigo3 does not use synthetic variable names. It automatically indents the code, which is necessary when variations introduce or remove *if* statements, and it eliminates blank lines due to empty tags. The file name of each generated program specifies the algorithm followed by all activated tags to make it easy to identify which file contains which code and what bugs are present, if any.

4.4 Subset selection

Combining the various implementation styles with all meaningful bug combinations yields 41,790 codes. Running them through a reasonable set of inputs results in millions of tests, which may take too long to run. To control the execution time, the suite supports the generation of user-defined subsets of the codes.

The code filtering is accomplished through a configuration file. We adopted this approach from Indigo [49] and chose it to simplify the subset creation. The configuration file lists the desired code versions and filters out the rest. For example, the user can elect to only generate bug-free codes. TACO [37] similarly creates tensor algebra kernels based on user-defined constraints. With this approach, an Indigo3 user can, for instance, generate a small subset for testing and later a more extensive subset to perform a detailed study.

The configuration file comprises 4 rules to manage the code generation as shown in Listing 15. The user can select the target graph algorithms, bug types, implementation styles, and data types. The example in Listing 15 generates every possible implementation style for all 7 graph algorithms, does not insert any bugs, and only uses the integer data type. The supported algorithms are breadth-first search (bfs), single-source shortest paths (sssp), connected components (cc), maximal independent set (mis), minimum spanning tree (mst), triangle counting (tc), and page rank (pr).

Table 5 lists the available choices for the code filters. As a shorthand, Indigo3 also supports the keywords “all” and “only”. The former turns off any filtering, and the latter means only code that includes the required tag will be generated. For example, putting “only RaceBug” in the bug option rule generates only the codes that have a race bug but do not include any other bugs.

```
CODE:
    algorithm:    {all}
    bug_option:   {nobug}
    style_option: {all}
    dataType:     {IntType}
```

Listing 15. Sample configuration file

Table 5. Choices for managing the code generation

Rule	Choices
Algorithm	all, bfs, sssp, cc, mis, mst, tc, pr
Bug option	all, nobug, bug names from Tables 2, 3, and 4
Style option	all, style names from Section 4.1
Data type	all, IntType, FloatType, LongType, DoubleType

5 EXPERIMENTAL METHODOLOGY

5.1 Hardware and software

The system we used for running the parallel C and OpenMP codes has two Intel Xeon Gold 6226R CPUs with 16 cores each. Hyperthreading is enabled, meaning the 32 cores can run 64 simultaneous threads. The main memory has a capacity of 128 GB. The operating system is Fedora 37. We ran the CUDA codes on an RTX 4090 GPU with 16,384 processing elements distributed over 128 multiprocessors. We compiled the CPU codes with *clang* 14.0.5 using the “-O3 -march=native” optimization flags, including “-fopenmp” for the OpenMP and “-pthread -std=c11” for the parallel C codes. We used *nvcc* 12.0.140 with the “-O3 -arch=sm_89” flags to compile the CUDA codes. We ran the CPU codes with 64 threads. For the CUDA experiments, we launched 512 threads per block.

5.2 Codes and inputs

Our test codes are based on 7 graph algorithms, namely Breadth-First Search, Connected Components, Single Source Shortest Path, Maximal Independent Set, Triangle Counting, PageRank, and Minimum Spanning Tree. We selected these algorithms because they are also frequently included in other benchmark suites. Since many existing program-analysis tools do not support the complex feature set of C++, we ported the Indigo2 C++ codes to C before including them in Indigo3. We generated the 2516 bug-free codes in the Indigo3 suite from these algorithms by applying the implementation and parallelization styles listed in Section 4. Since several of the code-verification tools we evaluated do not support the libcu++ library and parallel C, we removed the parallel C and CUDA codes that use this library from our tests, leaving 1924 bug-free codes. Half of them operate on 32-bit data types and the other half on 64-bit data types. To keep the running times manageable, we only evaluate the 32-bit data types in this paper.

To ensure compatibility with the iGuard [36] tool, we introduced the optional use of atomicAdd(0) and atomicExch for implementing atomic load and store operations in CUDA. Whereas these alternatives incur some performance overhead, they do broaden the range of tools to which our

Table 6. Graph information

Name	**Type**	**Origin**	**Vertices**	**Edges**	**Size (MB)**	d_{avg}	d_{max}	$d \geq 32$	$d \geq 512$	*Diameter*
soc-LiveJournal1	community	SNAP	4,847,571	85,702,474	362.2	17.7	20,333	14.0%	0.125%	21
rmat22.sym	RMAT	Galois	4,194,304	65,660,814	542.1	15.7	3,687	12.4%	0.045%	19
USA-road-d.NY	road map	Dimacs	264,346	730,100	6.9	2.8	8	0.0%	0.000%	721

codes can be applied. In summary, Indigo3 includes parallel C, OpenMP, and CUDA codes as well as alternative atomic load and store implementations for the CUDA tools that need it.

To thoroughly test the programs, we ran each of them on 67 input graphs, including one social network, one random graph, and one road map. Table 6 provides information on the type, size, and degree distribution of the three graphs. The remaining 64 inputs are all possible undirected graphs with four vertices. They are generated by enumerating all possible symmetric adjacency matrices.

5.3 Verification tools

We evaluate the effectiveness of 5 program-verification tools. Table 7 presents the type (static or dynamic), version, and the targeted programming model of each tool. Archer [1] is a data-race detector for OpenMP codes that combines static and dynamic techniques. ThreadSanitizer [8] is a dynamic data-race detector for C/C++ programs and is part of Clang 3.2 and gcc 4.8. We also tested CIVL [60], but being a static analyzer, it ended up being too slow to be included in our study.

iGUARD [36] instruments GPU programs to detect races in them. It is based on NVIDIA's NVBit binary instrumentation framework [65]. Compute Sanitizer [3] is a correctness-checking suite included in the CUDA toolkit. It contains multiple tools to perform different types of checks. The *memcheck* [5] tool detects out-of-bounds and misaligned memory accesses. It also reports hardware exceptions. The *racecheck* [6] tool flags shared memory data access hazards that can cause data races. The *initcheck* [4] tool checks for accesses to uninitialized data in global memory. The *synccheck* [7] tool reports cases where the application attempts invalid uses of synchronization primitives.

To accommodate the unique requirements of Archer and iGuard, which demand specific earlier versions of libraries and CUDA drivers, we implemented distinct setups to make them work. For Archer, we leveraged a Docker container environment, whereas iGuard is tested on a separate system with a Titan V GPU, CUDA driver version 418.39, and nvcc 10.1.

Table 7. Tested Verification Tools

Tool	**Type**	**Version**	**C/OpenMP**	**CUDA**
Clang Static Analyzer [2]	Static	18.0.0	Yes	No
Archer [1]	Dynamic/Static	2.0.0	Yes	No
ThreadSanitizer [8]	Dynamic	9.3.1	Yes	No
iGuard [36]	Dynamic	1.0	No	Yes
Compute Sanitizer [3]	Dynamic	2023.2.2	No	Yes

5.4 Metrics

To evaluate each tool, we measured the four counts shown in Table 8 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 8. 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:

$A = (TP + TN)/(TP + FP + TN + FN)$,

$P = TP/(TP + FP)$, and

$R = TP/(TP + FN)$.

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.

6 RESULTS

Applying all possible combinations of the 15 supported bug types to the 962 bug-free codes would result in hundreds of thousands of codes, and evaluating them on our 67 inputs would take many months. To make the running time manageable, we select four sets of codes for our experiments: (1) bug-free codes, (2) codes that have one parallelism bug, (3) codes that have one memory bug, and (4) codes that combine one parallelism bug. Additionally, we compare the generated bug-free codes with optimized third-party codes (i.e., Lonestar and Gardenia).

6.1 Bug-free codes

For the bug-free codes, if a tool reports a data race or memory bug, we count it as a false positive. Tables 9 and 10 list the tool, programming language, the number of evaluated codes, the number of these codes yielding a false positive for at least one input, the number of runs (i.e., codes × inputs), and the number of runs yielding a false positive. For example, ThreadSanitizer reports data races for 145 out of 12,596 runs, and these 145 runs stem from 4 bug-free codes.

Table 9 shows that Clang does not find any bugs in the bug-free CPU codes. Since it is a static analyzer that runs at compile time, it does not use any inputs. ThreadSanitizer reports non-existent data races in 4 codes, 2 of which use an OpenMP clause reduction and the other 2 swap two pointers to arrays after each iteration. Archer reports non-existent data races in 10 codes, all of which use an OpenMP clause reduction. Evidently, the internal implementation of the OpenMP reduction confuses both ThreandSanitizer and Archer. Additionally, ThreadSanitizer appears to not understand the implicit barrier at the end of a parallel code section, which is why swapping pointers between 2 such code sections yields false positives.

We made sure that the reported bugs are not actual bugs as follows. For the reduction problem, we changed the clause reduction to a critical section. With this change, ThreadSanitizer and Archer no longer output any data race warnings. For the swap problem, we duplicated the parallel code section and switched the array names in the second copy to eliminate the need for swapping the pointers. The modified code uses one copy in every odd iteration and the other copy in every even iteration. With this change, ThreadSanitizer no longer gives any data race warnings.

Table 10 shows that iGuard reports non-existent data races in 36 of the bug-free GPU codes, and Compute Sanitizer does not report any bugs. The false positives for iGuard stem from three scenarios:

codes that launch kernels at different granularities (e.g., thread-based and warp-based), codes that swap array pointers between kernels, and codes that access memory at different granularity (e.g., integer and Boolean arrays).

We modified the codes as follows to explore the reasons for the false positives and make sure they are not true positives. For the first scenario, we changed the kernels so that we could launch all of them at the same granularity. For the second condition, we first tried the idea outlined above to remove the swap. Since this did not help, we resorted to only launching 1 kernel at at time on the GPU and running the rest of the code on the CPU. For the third scenario, we converted the Boolean array into an integer array. These changes removed all iGuard data race reports. We believe the first two types of false positives arise because iGuard ignores the implicit barrier between kernel launches. The third type arises because we used iGuard's default memory-access granularity of 4 bytes, which is too coarse for Boolean arrays.

Table 9. Results for bug-free CPU codes

Tool	Language	Codes	FP Codes	Runs	FP Runs
Clang Static Analyzer	OpenMP	188	0 (0.0%)	n/a	n/a
ThreadSanitizer	OpenMP	188	4 (2.1%)	12,596	145 (1.2%)
Archer	OpenMP	188	10 (5.3%)	12,596	592 (4.7%)

Table 10. Results for bug-free GPU codes

Tool	Language	Codes	FP Codes	Runs	FP Runs
iGuard	CUDA	774	36 (4.6%)	51,858	1,974 (3.8%)
Compute Sanitizer	CUDA	774	0 (0.0%)	51,858	0 (0.0%)

6.2 Parallelism bug detection

Tables 11 and 12 show the results for the Indigo3 codes with exactly one parallelism bug. If a tool reports a data race or a missing barrier, we count it as a true positive result.

As Table 11 shows, the Clang Static Analyzer does not detect any of the bugs, presumably because it statically analyzes the program without considering inputs or runtime behavior. Both ThreadSanitizer and Archer detect some of the bugs, with ThreadSanitizer performing a little better. The GPU results in Table 12 show that both iGuard and Compute Sanitizer find a few of the bugs. iGuard performs better because Compute Sanitizer does not check for races in global memory.

The LivelockBug (see Table 2) is particularly challenging for ThreadSanitizer, Archer, and iGuard as evidenced by the increase in the percentages when removing the livelock codes. ThreadSanitizer correctly flags 118 (74.7%) and Archer 113 (71.5%) of 158 non-livelock buggy codes. iGuard correctly flags 201 (47.4%) of 424 non-livelock buggy codes. While iGuard has a timeout option, ThreadSanitizer and Archer potentially run forever if the program contains a livelock bug.

6.3 Memory bug detection

Since some memory bugs (e.g., out of bounds accesses) may cause data races, we count such reports as true positives. Tables 13 and 14 show the results for the codes with exactly one memory bug.

Even though the Clang Static Analyzer is not able to detect parallelism bugs, it does correctly report memory warnings for 19.1% of our codes. Archer detects more memory bugs and ThreadSanitizer even more, but both of them perform better on parallelism bugs than on memory bugs. This is

Table 11. Results for CPU codes with one parallelism bug

Tool	Language	Codes	TP Codes	Runs	TP Runs
Clang Static Analyzer	OpenMP	212	0 (0.0%)	n/a	n/a
ThreadSanitizer	OpenMP	212	136 (64.2%)	14,204	7,840 (55.2%)
Archer	OpenMP	212	115 (59.9%)	14,204	4,140 (31.9%)

Table 12. Results for GPU codes with one parallelism bug

Tool	Language	Codes	TP Codes	Runs	TP Runs
iGuard	CUDA	544	219 (40.3%)	36,448	10,326 (28.3%)
Compute Sanitizer	CUDA	544	53 (9.7%)	36,448	3,195 (8.8%)

not surprising because they are designed for data-race detection. On the GPU side, the same is true for iGuard. However, Compute Sanitizer performs much better on memory bugs. As mentioned, this is likely because it does not check for data races in global memory.

Table 13. Results for CPU codes with one memory bug

Tool	Language	Codes	TP Codes	Runs	TP Runs
Clang Static Analyzer	OpenMP	492	94 (19.1%)	n/a	n/a
ThreadSanitizer	OpenMP	492	276 (56.1%)	32,964	6,996 (22.2%)
Archer	OpenMP	492	160 (32.5%)	32,964	3,843 (11.7%)

Table 14. Results for GPU codes with one memory bug

Tool	Language	Codes	TP Codes	Runs	TP Runs
iGuard	CUDA	1,250	245 (19.6%)	83,750	12,363 (14.8%)
Compute Sanitizer	CUDA	1,250	765 (61.2%)	83,750	34,170 (40.8%)

6.4 Multiple bug detection

We also tested on Indigo3 codes with 2 bugs: 1 parallelism bug and 1 memory bug. Whenever a tool reports either a data race or a memory issue, we count it as a true positive. Tables 15 and 16 show the results for the codes with 2 bugs.

All evaluated tools perform better for the multiple-bug codes than for the single-bug codes. Similar to the single-bug results, ThreadSanitizer again finds more bugs than Archer. Compute Sanitizer reaches the highest true positives per code in all experiments as it detects many of the memory bugs and some data races trigger memory bugs that it can detect (e.g., races that write nonsensical values to a worklist).

Every tool generates incorrect predictions (false positives or false negatives). Section 6.1 discusses the reasons for false positives (i.e., when a tool reports bugs in bug-free codes). The reasons for false negatives (i.e., when a tool does not report an existing bug) are related to the design and implementation of the verification tools. For example, iGuard is a data race detection tool and not able to detect memory bugs. Additionally, some bugs (e.g., data races) may not manifest themselves in each run, making it difficult to detect for dynamic verifiers.

Table 15. Results for CPU codes with one memory and one parallelism bug

Tool	Language	Codes	TP Codes	Runs	TP Runs
Clang Static Analyzer	OpenMP	566	134 (24.0%)	n/a	n/a
ThreadSanitizer	OpenMP	566	443 (78.3%)	37,386	14,831 (39.7%)
Archer	OpenMP	566	430 (75.9%)	37,386	16,247 (43.5%)

Table 16. Results for GPU codes with one memory and one parallelism bug

Tool	Language	Codes	TP Codes	Runs	TP Runs
iGuard	CUDA	1,294	889 (68.7%)	86,698	40,835 (47.1%)
Compute Sanitizer	CUDA	1,294	1,097 (84.8%)	86,698	48,557 (56.0%)

6.5 Confusion matrix

Tables 17 and 18 evaluate the tools' effectiveness per code and per run, respectively. Higher numbers are better. For this study, we combined the inputs from the previous four subsections, that is, the bug-free codes, the codes with one parallelism bug, the codes with one memory bug, and the codes with both a parallelism and a memory bug. The results in Table 17 are higher than in Table 18 since bugs may not manifest themselves on every input. This illustrates the importance of thoroughly testing data-dependent codes on a range of inputs that elicit different runtime behaviors.

The precision is close to 100% in all cases, meaning the tools do not produce many false positives. Hence, if a tool reports a bug, it is likely that there is a true bug in the code. However, the highest accuracy and recall are below 72%, showing that the tools miss a substantial number of bugs.

ThreadSanitizer has a higher accuracy, precision, and recall than Archer. As discussed, Compute Sanitizer performs quite well even though it is unable to detect data races in global memory because, relatively speaking, it does very well at memory bug detection (and two of the three sets of buggy codes include memory errors).

Table 17. Tool metrics per code

Tool	Language	Accuracy	Precision	Recall
Clang Static Analyzer	OpenMP	28.5%	100.0%	18.0%
ThreadSanitizer	OpenMP	71.3%	99.5%	67.3%
Archer	OpenMP	61.1%	98.6%	56.1%
iGuard	CUDA	54.1%	97.4%	43.8%
Compute Sanitizer	CUDA	69.6%	100.0%	62.0%

6.6 Evaluation by style

The used parallelization and implementation style may impact the tools' effectiveness. To determine whether this is the case, we evaluate the tools on different styles. The results are shown in Tables 19, 20, 21, 22, and 23, where every row shows the metrics for a set of alternative styles.

In the following discussion, we focus on the most striking observations. For example, the Clang Static Analyzer finds more bugs in edge-based than in vertex-based codes. The opposite is true for ThreadSanitizer and Compute Sanitizer. A possible reason is that edge-based codes access the two endpoints of each edge, which may be simpler to analyze for a static tool than loops that iterate

Table 18. Tool metrics per run

Tool	Language	Accuracy	Precision	Recall
Clang Static Analyzer	OpenMP	n/a	n/a	n/a
ThreadSanitizer	OpenMP	43.1%	99.5%	34.9%
Archer	OpenMP	37.1%	97.6%	28.5%
iGuard	CUDA	43.8%	97.0%	30.7%
Compute Sanitizer	CUDA	53.2%	100.0%	41.5%

over variable-length adjacency lists as is done in vertex-based codes. Clang performs better on data-driven codes, but ThreadSanitizer and Archer detect more bugs in topology-driven codes, possibly because topology-driven codes exhibit more parallelism and, therefore, increase the chance of a parallelism bug manifesting itself. Archer and iGuard perform better for the pull than the push style. Since they are both data-race detectors, this may indicate that races in push-style codes are harder to detect. Perhaps the multiple-reader/multiple-writer races in the push style are more difficult to handle than the multiple-reader/single-writer races in the pull style. Furthermore, iGuard detects more bugs for the non-duplicate worklist and read-write styles than their alternatives. One reason may be that read-write versions have independent read and write operations, which increases the chance for a data race. Averaged over all tested tools, programs implemented in the data-driven and pull styles tend to be the easiest to verify, and programs that allow duplicates on the worklist are the most challenging. Overall, we find that the verification tools perform differently on alternative styles. This highlights the importance of thoroughly testing and evaluating verification tools using programs that are implemented in different styles.

Table 19. Clang's evaluation for each style

Tool	Accuracy	Precision	Recall
Vertex, Edge	26%, 42%	100%, 100%	14%, 32%
Topo, Data	20%, 43%	100%, 100%	2%, 35%
NonDup, Dup	24%, 33%	100%, 100%	12%, 22%
Push, Pull	25%, 23%	100%, 100%	14%, 14%
ReadWrite, ReadModifyWrite	24%, 26%	100%, 100%	17%, 19%
NonDeterm, Determ	25%, 33%	100%, 100%	15%, 22%
Default, Dynamic	30%, 30%	100%, 100%	19%, 19%
AtomicAdd, CriticalAdd, ClauseAdd	25%, 25%, 25%	100%, 100%, 100%	13%, 13%, 13%

6.7 Comparison with third-party codes

To demonstrate that our unoptimized bug-free codes yield reasonable performance, we compare them to the optimized Lonestar [39] CPU and Gardenia [70] GPU codes. We refer to these Lonestar and Gardenia codes as "baseline". We omitted some of the modifications to our codes described in Section 5.2 since they merely serve to make the codes compatible with the verification tools. For each of our codes in this analysis, we selected the style that yields the highest average throughput across all inputs. Then we run the best-performing style on the set of inputs listed in Table 24. We selected them because they cover a wide range of sizes and degree distributions.

We compute the speedups over the baseline codes and visualize them in Figure 2. Each column summarizes the speedups over all inputs for one algorithm. Since we run each program through a

Table 20. ThreadSanitizer's evaluation for each style

Tool	Accuracy	Precision	Recall
Vertex, Edge	76%, 42%	99%, 98%	71%, 31%
Topo, Data	83%, 69%	99%, 100%	79%, 64%
NonDup, Dup	71%, 67%	100%, 100%	65%, 60%
Push, Pull	78%, 75%	99%, 100%	75%, 74%
ReadWrite, ReadModifyWrite	71%, 71%	100%, 99%	67%, 69%
NonDeterm, Determ	75%, 77%	100%, 99%	71%, 73%
Default, Dynamic	80%, 73%	99%, 99%	77%, 69%
AtomicAdd, CriticalAdd, ClauseAdd	86%, 86%, 80%	100%, 100%, 96%	84%, 84%, 81%

Table 21. Archer's evaluation for each style

Tool	Accuracy	Precision	Recall
Vertex, Edge	59%, 62%	98%, 99%	52%, 55%
Topo, Data	66%, 54%	96%, 100%	60%, 47%
NonDup, Dup	47%, 46%	100%, 100%	37%, 35%
Push, Pull	55%, 64%	99%, 98%	48%, 60%
ReadWrite, ReadModifyWrite	47%, 55%	100%, 100%	40%, 51%
NonDeterm, Determ	58%, 60%	99%, 98%	52%, 53%
Default, Dynamic	69%, 54%	99%, 98%	64%, 47%
AtomicAdd, CriticalAdd, ClauseAdd	64%, 68%, 78%	100%, 100%, 85%	58%, 63%, 90%

Table 22. iGuard's evaluation for each style

Tool	Accuracy	Precision	Recall
Vertex, Edge	51%, 44%	98%, 100%	41%, 39%
Topo, Data	53%, 55%	100%, 95%	41%, 41%
NonDup, Dup	40%, 29%	100%, 100%	27%, 5%
Push, Pull	44%, 56%	93%, 93%	28%, 53%
ReadWrite, ReadModifyWrite	43%, 34%	100%, 100%	34%, 8%
NonDeterm, Determ	50%, 54%	94%, 93%	41%, 46%
Persist, NonPersist	46%, 49%	94%, 94%	39%, 43%
Thread, Warp, Block	34%, 42%, 55%	97%, 95% 93%	20%, 35%, 53%
GlobalAdd, BlockAdd, ReductionAdd	49%, 39%, 39%	92%, 92% 92%	48%, 38%, 38%

set of inputs, each column represents multiple speedups. The box shows the range of the middle 50% of the data. The line in the middle of the box indicates the median. Other data points are plotted as circles. Speedups above 1 (i.e., the dashed blue line) mean our codes are faster. If the median line in the box is above 1, it shows that our codes are faster than the baseline for at least half of the inputs. Figure 2a does not show MIS or MST results since they are not included in Gardenia [70].

Our PR and TC codes outperform the CPU baselines but are slower on the GPUs because the Gardenia codes include an optimization that removes redundant edges. The performance of CC is on par with the baselines across the different devices and programming models. Our BFS codes are

Table 23. ComputeSanitizer's evaluation for each style

Tool	Accuracy	Precision	Recall
Vertex, Edge	71%, 57%	100%, 100%	64%, 53%
Topo, Data	70%, 74%	100%, 100%	62%, 65%
NonDup, Dup	71%, 67%	100%, 100%	64%, 55%
Push, Pull	68%, 65%	100%, 100%	57%, 59%
ReadWrite, ReadModifyWrite	63%, 69%	100%, 100%	57%, 57%
NonDeterm, Determ	67%, 69%	100%, 100%	60%, 61%
Persist, NonPersist	68%, 69%	100%, 100%	62%, 64%
Thread, Warp, Block	68%, 68%, 67%	100%, 100% 100%	60%, 63%, 63%
GlobalAdd, BlockAdd, ReductionAdd	66%, 63%, 64%	100%, 100% 100%	63%, 60%, 61%

Table 24. Inputs for performance comparison

Name	Type	Origin	Vertices	Edges	Size (MB)
2d-2e20.sym	grid	Galois	1,048,576	4,190,208	37.7
coPapersDBLP	publication	SMC	540,486	30,491,458	124.1
rmat22.sym	RMAT	Galois	4,194,304	65,660,814	542.1
soc-LiveJournal1	community	SNAP	4,847,571	85,702,474	362.2
USA-road-d.NY	road map	Dimacs	264,346	730,100	6.9

Table 25. Average speedup over baseline codes

Language	BFS	SSSP	CC	MIS	PR	TC	Geomean
CUDA	1.97	0.40	1.11	N/A	0.45	0.43	0.70
OpenMP	0.90	0.10	0.89	6.55	2.86	5.11	1.54
C++ threads	1.14	0.07	0.51	21.14	12.47	3.04	1.80

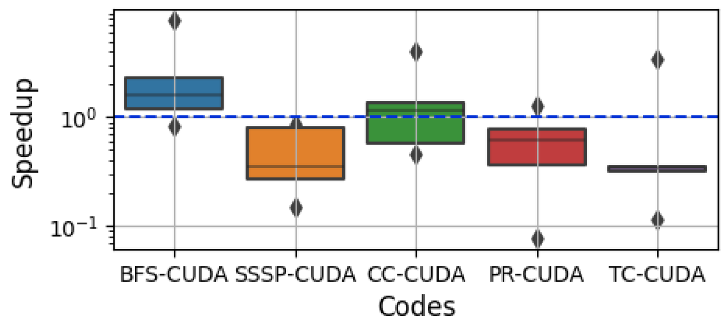
faster on the GPUs and similar to the baseline on the CPUs. Lastly, our SSSP codes are generally slower. This is because both Lonestar and Gardenia include worklist optimizations. Gardenia employs two extra arrays that make the code as efficient as the data-driven approach but without the overhead of maintaining a worklist. Lonestar combines the data-driven approach with a priority scheduler that processes the vertices in ascending distance to reduce the total amount of work.

Table 25 lists the average speedup of the best-performing style over the baseline for each algorithm. For example, the "1.97" in the CUDA row and BFS column means our BFS CUDA code is 1.97× faster on average (i.e., geometric mean). The right-most column presents the geometric mean for each programming model.

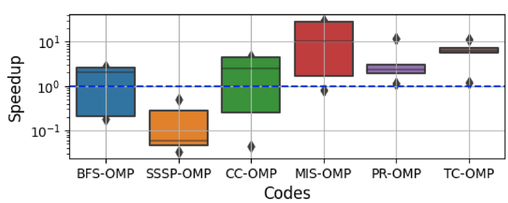
Overall, we find that, even though our codes do not include optimizations, they still yield reasonable performance. The optimized baselines do not outperform our codes in many cases, indicating that choosing the right implementation style is as important as incorporating program-specific code optimizations.

6.8 Result correlation with inputs and architectures

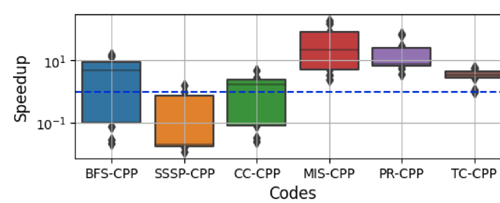
As the behavior of our codes is input and hardware dependent, we studied the results for each input graph on different devices. We found that the degree distribution (e.g., road maps versus social networks) does not significantly influence the results. However, the graph size can impact the data race detection on the CPU. For example, ThreadSanitizer detects more data races in larger graphs (73% of the parallelism bugs) than in smaller graphs (62%). The larger graphs include the



(a) CUDA



(b) OpenMP



(c) C++ Threads

Fig. 2. Throughput ratio to baseline codes

all-possible 4-vertex graphs with more than 3 edges as well as the real-world graphs. The smaller graphs are the 4-vertex graphs with 3 or fewer edges. In contrast, the CUDA tools perform the same across different graph sizes. Hence, the behavior for different programming models (i.e., OpenMP and CUDA) can be different. Moreover, we found that a tool may produce a different prediction for the same program on different hardware. This is expected because dynamic tools often yield different reports for each run anyways. However, since we run a large number of tests, the overall results for a specific tool tend to be consistent across different hardware.

7 SUMMARY AND CONCLUSIONS

This paper presents a labeled benchmark suite called Indigo3 [46] that includes 41,790 graph analytics codes written in CUDA, OpenMP, and parallel C. Each program can be run with an unbounded number of inputs. They are based on 13 sets of alternative parallelization/implementation styles and 15 types of common bugs. We wrote a framework to automatically create the Indigo3 suite by generating codes with all meaningful combinations of these styles and bugs as well as bug-free codes. We applied our framework to 7 graph algorithms expressed in 3 programming models. Each generated code is labeled with the parallelization/implementation styles and bugs present. This allows users to select desired subsets and makes Indigo3 useful for testing various tools.

We evaluated 5 program verification tools on 4 subsets of Indigo3 codes, namely codes that are bug-free, have one parallelism bug, have one memory bug, and combine one parallelism with one memory bug. The results show that ThreadSanitizer, Archer, and iGuard are better at detecting parallelism bugs whereas the Clang Static Checker and Computer Sanitizer are better at detecting memory bugs. Since memory bugs may manifest themselves as data races, data-race warnings are sometimes triggered by memory bugs. We carefully examined all reported false positives to make sure our bug-free codes are correct and to determine the program patterns that confuse the verifiers. The results per code are always significantly better than per input, meaning data-dependent codes such as irregular graph algorithms should be tested on a number of inputs that elicit different program behaviors. Additionally, we found the tools' effectiveness to vary between implementation styles, highlighting the importance of considering different styles when testing verification tools. We hope our work will prove useful to the verification community and will inspire others to build benchmark suites for additional domains.

ACKNOWLEDGMENTS

This work has been supported in part by the National Science Foundation under Award Number 1955367 and by an equipment donation from NVIDIA Corporation. We thank Ganesh Gopalakrishnan, John Jacobson, Stephen Siegel, Alex Wilton, and Wenhao Wu for their help and feedback to improve this paper.

REFERENCES

[1] Accessed: 2021-6-26. Archer. https://github.com/PRUNERS/archer.
[2] Accessed: 2023-9-28. Clang Static Analyzer. https://clang-analyzer.llvm.org/.
[3] Accessed: 2023-9-28. ComputeSanitizer. https://docs.nvidia.com/compute-sanitizer/index.html.
[4] Accessed: 2023-9-28. Initcheck Tool. https://docs.nvidia.com/compute-sanitizer/ComputeSanitizer/index.html.
[5] Accessed: 2023-9-28. Memcheck Tool. https://docs.nvidia.com/compute-sanitizer/ComputeSanitizer/index.html.
[6] Accessed: 2023-9-28. Racecheck Tool. https://docs.nvidia.com/compute-sanitizer/ComputeSanitizer/index.html.
[7] Accessed: 2023-9-28. Synccheck Tool. https://docs.nvidia.com/compute-sanitizer/ComputeSanitizer/index.html.
[8] Accessed: 2023-9-28. ThreadSanitizer. https://github.com/google/sanitizers.
[9] Lada A Adamic, Rajan M Lukose, Amit R Puniyani, and Bernardo A Huberman. 2001. Search in power-law networks. *Physical review E* 64, 4 (2001), 046135.
[10] Gabriell Araujo, Dalvan Griebler, Dinei A. Rockenbach, Marco Danelutto, and Luiz G. Fernandes. 2021. NAS Parallel Benchmarks with CUDA and beyond. *Software: Practice and Experience* 53, 1 (2021), 53–80. https://doi.org/10.1002/spe.3056 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.3056
[11] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. 2012. Efficient system-enforced deterministic parallelism. *Commun. ACM* 55, 5 (2012), 111–119.
[12] Scott Beamer, Krste Asanovic, and David Patterson. 2012. Direction-optimizing breadth-first search. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, IEEE, New York, NY, USA, 1–10.
[13] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).
[14] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 2017. To push or to pull: On reducing communication and synchronization in graph computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. 93–104.
[15] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. 2012. GPUVerify: a verifier for GPU kernels. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 113–132.
[16] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. 72–81.
[17] Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, and Julian Shun. 2012. Internally deterministic parallel algorithms can be fast. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. 181–192.
[18] Hans-J. Boehm. 2011. How to Miscompile Programs with "Benign" Data Races. In *3rd USENIX Workshop on Hot Topics in Parallelism (HotPar 11)*. USENIX Association, Berkeley, CA. https://www.usenix.org/conference/hotpar-11/how-miscompile-programs-benign-data-races
[19] Hans-J Boehm. 2011. How to miscompile programs with" benign" data races. In *3rd USENIX Workshop on Hot Topics in Parallelism (HotPar 11)*.
[20] Martin Burtscher. 2019. ECL graphs. https://cs.txstate.edu/~burtscher/research/ECLgraph/index.html. Accessed: 2023-08-18.
[21] Martin Burtscher and Jared Coplin. 2014. Power characteristics of irregular GPGPU programs. In *Workshop on General Purpose Processing Using GPUs*.

[22] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. 2012. A quantitative study of irregular programs on GPUs. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 141–151.

[23] Shuai Che, Bradford M Beckmann, Steven K Reinhardt, and Kevin Skadron. 2013. Pannotia: Understanding irregular GPGPU graph applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 185–195.

[24] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 44–54.

[25] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. 2010. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. 63–74.

[26] Laxman Dhulipala, Jessica Shi, Tom Tseng, Guy E Blelloch, and Julian Shun. 2020. 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)*. 1–8.

[27] Jack Dongarra. [n. d.]. Compressed row storage. http://www.netlib.org/utk/people/JackDongarra/etemplates/node373.html. Accessed: 2021-7-3.

[28] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J Boehm. 2012. IFRit: interference-free regions for dynamic data-race detection. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 467–484.

[29] Ariel Eizenberg, Yuanfeng Peng, Toma Pigli, William Mansky, and Joseph Devietti. 2017. Barracuda: Binary-level analysis of runtime races in cuda programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 126–140.

[30] Adriano Marques Garcia, Dalvan Griebler, Claudio Schepke, and Luiz Gustavo Fernandes. 2022. SPBench: a framework for creating benchmarks of stream processing applications. *Computing* 105, 5 (jan 2022), 1077–1099. https://doi.org/10.1007/s00607-021-01025-6

[31] Alan George, Joseph WH Liu, et al. 1981. *Computer solution of large sparse positive definite systems*. Vol. 134. Prentice-Hall Englewood Cliffs, NJ.

[32] Juan Gómez-Luna, Izzat El Hajj, Li-Wen Chang, Víctor García-Floreszx, Simon Garcia De Gonzalo, Thomas B Jablin, Antonio J Pena, and Wen-mei Hwu. 2017. Chai: Collaborative heterogeneous applications for integrated-architectures. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 43–54.

[33] Dalvan Griebler, Marco Danelutto, Massimo Torquati, and Luiz Gustavo Fernandes. 2017. SPar: A DSL for High-Level and Productive Stream Parallelism. *Parallel Processing Letters* 27, 01 (2017), 1740005. https://doi.org/10.1142/S0129626417400059

[34] Kshitij Gupta, Jeff A. Stuart, and John D. Owens. 2012. A study of Persistent Threads style GPU programming for GPGPU workloads. In *2012 Innovative Parallel Computing (InPar)*. IEEE, San Jose, CA, USA, 1–14. https://doi.org/10.1109/InPar.2012.6339596

[35] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. 2011. Accelerating CUDA Graph Algorithms at Maximum Warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming* (San Antonio, TX, USA) *(PPoPP '11)*. Association for Computing Machinery, New York, NY, USA, 267–276. https://doi.org/10.1145/1941553.1941590

[36] Aditya K. Kamath and Arkaprava Basu. 2021. IGUARD: In-GPU Advanced Race Detection. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 49–65. https://doi.org/10.1145/3477132.3483545

[37] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (Oct. 2017), 29 pages. https://doi.org/10.1145/3133901

[38] Milind Kulkarni, Martin Burtscher, Calin Casçaval, and Keshav Pingali. 2009. Lonestar: A suite of parallel irregular programs. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 65–76.

[39] Milind Kulkarni, Martin Burtscher, Calin Cascaval, and Keshav Pingali. 2009. Lonestar: A suite of parallel irregular programs. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, New York, NY, USA, 65–76. https://doi.org/10.1109/ISPASS.2009.4919639

[40] Gary T Leavens. 2007. The java modeling language (jml). *URL http://sourceforge. net/apps/wordpress/fixedptc* (2007).

[41] Da Li, Hancheng Wu, and Michela Becchi. 2015. Nested Parallelism on GPU: Exploring Parallelization Templates for Irregular Loops and Recursive Computations. In *2015 44th International Conference on Parallel Processing*. IEEE, New York, NY, USA, 979–988. https://doi.org/10.1109/ICPP.2015.107

[42] Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P. Rajan. 2012. GKLEE: Concolic Verification and Test Generation for GPUs. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New Orleans, Louisiana, USA) *(PPoPP '12)*. Association for Computing Machinery, New York, NY, USA, 215–224. https://doi.org/10.1145/2145816.2145844

[43] Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan, and Ian Karlin. 2017. 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*. 1–14.

[44] Richard J Lipton. 1975. Reduction: A method of proving properties of parallel programs. *Commun. ACM* 18, 12 (1975), 717–721.

[45] Yiqian Liu, Noushin Azami, Avery Vanausdal, and Martin Burtscher. 2023. Choosing the Best Parallelization and Implementation Styles for Graph Analytics Codes: Lessons Learned from 1106 Programs. In *SC '23: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 11111. https://doi.org/11111

[46] Yiqian Liu, Noushin Azami, Avery VanAusdal, and Martin Burtscher. 2023. Indigo3 Git Repository. https://github.com/burtscher/Indigo3Suite. Accessed: 2023-12-09.

[47] Yiqian Liu, Noushin Azami, Corbin Walters, and Martin Burtscher. 2022. The Indigo Program-Verification Microbenchmark Suite of Irregular Parallel Code Patterns. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 24–34. https://doi.org/10.1109/ISPASS55109.2022.00003

[48] Yiqian Liu, Noushin Azami, Corbin Walters, and Martin Burtscher. 2022. The Indigo Program-Verification Microbenchmark Suite of Irregular Parallel Code Patterns. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 24–34.

[49] Yiqian Liu, Noushin Azami, Corbin Walters, and Martin Burtscher. 2022. The Indigo Program-Verification Microbenchmark Suite of Irregular Parallel Code Patterns. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, New York, NY, USA, 24–34. https://doi.org/10.1109/ISPASS55109.2022.00003

[50] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and Ching-Yung Lin. 2015. 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*. 1–12. https://doi.org/10.1145/2807591.2807626

[51] Masahiro Nakao, Koji Ueno, Katsuki Fujisawa, Yuetsu Kodama, and Mitsuhisa Sato. 2020. Performance Evaluation of Supercomputer Fugaku using Breadth-First Search Benchmark in Graph500. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. 408–409. https://doi.org/10.1109/CLUSTER49012.2020.00053

[52] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Atomic-Free Irregular Computations on GPUs. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units* (Houston, Texas, USA) *(GPGPU-6)*. Association for Computing Machinery, New York, NY, USA, 96–107. https://doi.org/10.1145/2458523.2458533

[53] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Atomic-free irregular computations on GPUs. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. 96–107.

[54] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Data-Driven Versus Topology-driven Irregular Computations on GPUs. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, New York, NY, USA, 463–474. https://doi.org/10.1109/IPDPS.2013.28

[55] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Data-driven versus topology-driven irregular computations on GPUs. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 463–474.

[56] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Morph Algorithms on GPUs. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP '13)*. Association for Computing Machinery, New York, NY, USA, 147–156. https://doi.org/10.1145/2442516.2442531

[57] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, et al. 2011. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 12–25.

[58] Apan Qasem, Ashwin M. Aji, and Gregory Rodgers. 2017. Characterizing data organization effects on heterogeneous memory architectures. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 160–170. https://doi.org/10.1109/CGO.2017.7863737

[59] Simon Schwitanski, Joachim Jenke, Sven Klotz, and Matthias S. Müller. 2023. RMARaceBench: A Microbenchmark Suite to Evaluate Race Detection Tools for RMA Programs *(SC-W '23)*. Association for Computing Machinery, New York, NY, USA, 205–214. https://doi.org/10.1145/3624062.3624087

[60] Stephen F. Siegel, Manchun Zheng, Ziqing Luo, Timothy K. Zirkel, Andre V. Marianiello, John G. Edenhofner, Matthew B. Dwyer, and Michael S. Rogers. 2015. CIVL: the concurrency intermediate verification language. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1145/2807591.2807635

[61] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012).

[62] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. 2019. Understanding real-world concurrency bugs in Go. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 865–878.

[63] Swapneela Unkule, Christopher Shaltz, and Apan Qasem. 2012. Automatic restructuring of GPU kernels for exploiting inter-thread data locality. In *International Conference on Compiler Construction*. Springer, 21–40.

[64] Gaurav Verma, Yaying Shi, Chunhua Liao, Barbara Chapman, and Yonghong Yan. 2020. Enhancing DataRaceBench for Evaluating Data Race Detection Tools. In *2020 IEEE/ACM 4th International Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE, 20–30.

[65] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W. Keckler. 2019. NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 372–383. https://doi.org/10.1145/3352460.3358307

[66] Jin Wang and Sudhakar Yalamanchili. 2014. Characterization and analysis of dynamic parallelism in unstructured GPU applications. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, New York, NY, USA, 51–60. https://doi.org/10.1109/IISWC.2014.6983039

[67] Hancheng Wu, Da Li, and Michela Becchi. 2016. Compiler-Assisted Workload Consolidation for Efficient Dynamic Parallelism on GPU. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, New York, NY, USA, 534–543. https://doi.org/10.1109/IPDPS.2016.98

[68] Mingyuan Wu, Yicheng Ouyang, Husheng Zhou, Lingming Zhang, Cong Liu, and Yuqun Zhang. 2020. Simulee: Detecting CUDA Synchronization Bugs via Memory-Access Modeling. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 937–948. https://doi.org/10.1145/3377811.3380358

[69] Zhen Xu, Xuhao Chen, Jie Shen, Yang Zhang, Cheng Chen, and Canqun Yang. 2019. Gardenia: A graph processing benchmark suite for next-generation accelerators. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 15, 1 (2019), 1–13.

[70] Zhen Xu, Xuhao Chen, Jie Shen, Yang Zhang, Cheng Chen, and Canqun Yang. 2019. GARDENIA: A Graph Processing Benchmark Suite for Next-Generation Accelerators. *J. Emerg. Technol. Comput. Syst.* 15, 1, Article 9 (jan 2019), 13 pages. https://doi.org/10.1145/3283450

[71] Ting Yuan, Guangwei Li, Jie Lu, Chen Liu, Lian Li, and Jingling Xue. 2021. GoBench: A Benchmark Suite of Real-World Go Concurrency Bugs. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 187–199.

[72] P Zhang and G Chartrand. 2006. *Introduction to graph theory*. Tata McGraw-Hill.

[73] Yang Zhang, Jie Shen, Zhen Xu, Shikai Qiu, and Xuhao Chen. 2019. Architectural Implications in Graph Processing of Accelerator with Gardenia Benchmark Suite. In *2019 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*. IEEE, 1329–1339.

Received 10 December 2023; revised 8 May 2024; accepted 7 May 2024