Performance Impact of Removing Data Races from GPU Graph Analytics Programs

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

Avery VanAusdal
Department of Computer Science
Texas State University
San Marcos, TX, USA
arv107@txstate.edu

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

***Abstract*—Some of the fastest CUDA codes contain "benign" data races to boost their performance. However, such races can lead to unpredictable behavior and incorrect results on other hardware and compilers, making their elimination crucial for producing reliable and portable programs. This paper investigates the performance impact of removing data races from six high-end graph analytics codes. We identify and eliminate the races from these GPU programs by adding necessary synchronization and validating their correctness. We present our race-free codes and their original versions as an open-source suite. Comparing the performance of our new codes with their baseline counterparts on multiple inputs and GPUs, we observe that race-free implementations do not always incur a performance penalty. In fact, some race-free versions are faster, with our validated maximal independent set implementation achieving a 5-11% speedup. Our findings indicate that race-free code can reach comparable or even superior performance, supporting the adoption of best practices for parallel programming.**

***Index Terms*—Parallel programming, data races, software verification and validation, graph analytics, CUDA**

I. Introduction

Data analytics has become a major workload in recent years. Due to the high computational demands, GPUs are often used for running such analytics. In many cases, the inputs consist of semi- or unstructured data, requiring irregular algorithms [1] for processing them that can be difficult to parallelize and optimize, especially for GPUs.

To maximize performance and simplify implementation, programmers sometimes resort to tricks such as using "benign" data races that do not affect program correctness on their system. In fact, CUDA explicitly allows such races via the *volatile* qualifier (see below). However, these tricks yield non-portable programs that may not work on future generations of the same GPU family or on other current GPU families. Moreover, they may result in miscompiled code when switching to a different compiler or to a future version of the same compiler [2]. For these reasons, many multithreading specifications, including those for C [3], C++ [4], Posix threads [5], OpenMP [6], and now CUDA [7], specify all data races as undefined behavior. Consequently, any source code with a data race is incorrect. In other words, there is no such thing as a "benign" data race in high-level programming languages [2].

Current CUDA-capable GPUs load and store scalar values in a single access. Hence, it is common for programmers to not protect memory operations to shared scalar data beyond labelling the data as *volatile*. The *volatile* qualifier alerts the compiler that a memory location may be concurrently used or changed by other threads, so all accesses to it must compile into an actual memory read or write instruction without local caching [7]. However, the assumption that these accesses are atomic may not hold on other architectures or future generations, breaking previously working code. Therefore, the *volatile* keyword does not solve the portability issue and disables optimizations.

As a remedy, NVIDIA recently introduced *libcu++*, a C++ Standard Library that can be used both in and between CPU and GPU code [8]. This library allows the declaration of atomic data types with optional memory-ordering and scope specifications, which were not available for atomic operations in CUDA before. The memory order restricts how the surrounding memory accesses can be reordered with respect to the atomic operation. The scope determines whether the operation is atomic at the block, grid, or system level (including host code). Whereas *libcu++* is a welcome addition, it does not support older GPUs and is new enough that most codes do not use it yet. Moreover, it places an additional burden on the programmer, who may need to determine the appropriate memory ordering and scope, especially given that the defaults can lead to poor performance [9].

In "regular" codes, such as those that operate on dense matrices and arrays, data races can often easily be avoided even without synchronization. This is not the case for "irregular" codes with unpredictable memory-access patterns and control flow, of which graph analytics are an important and widely used representative.

To investigate the prevalence and performance benefit of

This work has been supported in part by the National Science Foundation under Award Number 1955367 and by an equipment donation from NVIDIA Corporation.

"benign" data races, we studied six high-performance graph analytics CUDA codes: all-pairs shortest-paths (APSP) [10], connected components (CC) [11], graph coloring (GC) [12], maximal independent sets (MIS) [13], minimum spanning trees (MST) [14], and strongly connected components (SCC) [15]. We selected these codes because they are some of the fastest GPU implementations of their respective graph algorithms in the current literature.

We removed all data races from these codes using various techniques as explained in Section IV. We validated our modified codes with multiple verification tools (and by hand) to ensure that they are data-race free. Then, we studied the performance impact of removing the "benign" data races on different inputs and GPU generations. Surprisingly, our results show that the race-free codes are not necessarily slower. Whereas our modifications make some codes between 1.1 and 2.2 times slower, our race-free MIS implementation is around 5-11% faster than its baseline counterpart. This likely makes it the currently fastest CUDA implementation of MIS. In general, we found both the algorithm implementation and the input size to impact the performance when removing data races from a program. This paper makes the following main contributions.

- It identifies latent data races in several high-performance graph analytics CUDA codes.
- It explains how we modified these codes to make them data-race free and how we verified them.
- It compares the execution time of our new codes with their baselines to quantify the performance impact of making programs data-race free.
- It demonstrates that removing "benign" races can, in fact, improve performance and analyzes the conditions under which this might happen.

Both the baseline and validated race-free codes are open-sourced and freely available in our ECL-Suite [16].

The rest of the paper is organized as follows. Section II reviews relevant background information and introduces the six studied codes. Section III summarises related work. Section IV describes our approach for identifying and removing data races. Section V discusses the experimental methodology. Section VI evaluates the performance impact of removing data races from graph analytics codes. Section VII summarizes our findings and draws conclusions.

II. Background

A. Concurrency Issues

This subsection reviews synchronization-related aspects of parallel programs that are important for our study.

- **Atomic access**: a load or store memory operation that cannot be interrupted.
- **Word tearing**: a single-element access that is performed using multiple loads or stores.
- **Volatile variable**: a memory location that can change its value (e.g., due to another thread).
- **Memory ordering**: restrictions on which memory accesses can be reordered relative to each other.

Word tearing happens, for example, when a 64-bit variable is read or written on a 32-bit architecture that does not have machine instructions for directly accessing 64-bit words. On such a system, a 64-bit write is accomplished by two 32-bit stores and a 64-bit load by two 32-bit loads. To see why this is a problem, consider Thread T_2 running concurrently with T_1 in Fig. 1. If word tearing occurs, T_2 will not always print -1 or 0 as one might expect but may print the chimera value 0xffffffff00000000 or 0x00000000ffffffff, in which half of the bits come from the initialization value -1 and the other half from the value 0 that T_1 writes. This happens when the thread timing is such that T_1 executes its first 32-bit store, then T_2 executes its two 32-bit loads (to fetch the value for printing), and finally T_1 executes its second 32-bit store.

Word tearing is a problem even when another thread performs an atomic access, as is the case for Thread T_3 in Fig. 1, which atomically adds 6 to *val*. The atomic add executes a read-modify-write operation, that is, it both loads and stores *val* atomically. Note that the entire operation is a single atomic transaction, it does not perform an atomic load followed by a separate atomic store. Thus, other threads can only access *val* before the atomic add starts or after it has completed, but not after having loaded *val* and before storing the updated result to *val*. Nevertheless, word tearing in T_1 would still be a problem because the atomic add might execute after T_1 has set the top half of the bits of *val* to 0 but not yet the bottom half. This yields the following three possibilities. If T_1 runs before T_3, the final value of *val* will be 6. If T_3 runs before T_1, the final value will be 0. If T_1 and T_3 interleave due to word tearing in T_1, the final value will be the nonsensical 0x0000000100000000.

In general, using non-atomic accesses results in *unsafe* parallel programs even if word tearing never occurs on the target machine, for example, because it supports 64-bit loads and stores that are natively atomic. Such programs are still unsafe because they are not portable. If they are compiled and executed on a machine that does not support 64-bit accesses directly, they may fail.

Another potential problem is illustrated by Thread T_4. Interestingly, it is not affected by the potential word tearing of T_1 because it only checks whether *val* has changed but not to which value it changed (i.e., a chimera value is acceptable). Nevertheless, the T_4 code is wrong as it might result in an infinite loop. Since the access to *val* is performed using a conventional load, there is no indication that some other thread might change this value. Hence, the compiler could move the (seemingly) loop-invariant load out of the loop and register allocate the *data* variable. Once the load has executed, changing *val* by another thread will not change the value in the *data* register and the loop in T_4 will never terminate. The keyword *volatile* in CUDA indicates that other threads might change the corresponding memory location at

```
Shared variable      Thread T1    Thread T2             Thread T3               Thread T4
---------------      ---------    ---------             ---------               ---------
long val = -1;       val = 0;     printf("%ld", val);   atomicAdd(&val, 6);     do {
                                                                                   data = val;
                                                                                } while (data == -1);
                                                                                printf("value changed");
```

Fig. 1: Initial state of a shared variable *val* and the instructions executed by four threads

any time, preventing the compiler from performing this and some other optimizations. However, marking a variable as *volatile* does not prevent word tearing, which is why atomic accesses should be used rather than volatile variables.

The atomic operations in *libcu++* support multiple memory ordering restrictions. *Relaxed* is the weakest (all reordering allowed), *seq_cst* is the strongest (no reordering allowed), and *acquire* and *release* are in between (some reordering allowed) [4]. The weakest version that is sufficient for correctness should be used to maximize performance. In all our codes, we are able to use the *relaxed* memory ordering.

B. Included Codes

This subsection describes the baseline codes we evaluated and modified for our study.

1) All pairs shortest paths: All-pairs-shortest-paths (APSP) algorithms calculate the shortest distance between every pair of vertices in weighted graphs. The ECL-APSP code [10] is based on the blocked approach of the Floyd-Warshall algorithm [17]. It divides the adjacency matrix into 64×64-element subblocks, and each kernel launch processes multiple iterations on all subblocks. Utilizing the shared memory on the GPU and maximizing the subblock size significantly reduces global memory accesses. We found ECL-APSP to be 1.52× faster on average than the next fastest code [18].

2) Connected components: A connected component (CC) is a set of vertices in an undirected graph that are linked to each other by paths. The ECL-CC code [11] is based on the label propagation technique. It employs a GPU-friendly union-find method and, to improve the load balance, processes the vertices at thread, warp, or block granularity depending on the number of neighbors. Additionally, the code is asynchronous and lock-free. ECL-CC is 1.8× faster than the fastest prior CC implementation for GPUs [19].

3) Graph coloring: Graph-coloring (GC) algorithms assign colors to the vertices of an undirected graph such that adjacent vertices use different colors, while minimizing the number of distinct colors used. The ECL-GC code [12] builds upon the Jones-Plassmann algorithm [20] with the largest-degree-first heuristic. It includes two shortcut optimizations. The first shortcut identifies opportunities to already color lower-priority vertices when their higher-priority neighbors are still uncolored, which increases parallelism. The second shortcut reduces the number of colors to consider when a vertex has a higher-priority neighbor with no overlap in the possible colors. ECL-GC is 2.9× faster and uses as few or fewer colors as the best prior GPU code for GC [21].

4) Maximal independent set: A maximal independent set (MIS) is a subset of vertices of an undirected graph that contains no adjacent vertices and to which no other vertices can be added without introducing adjacent vertices in the set. The ECL-MIS code [13] builds upon Luby's algorithm [22]. It employs several optimizations to improve the speed of the computation as well as the size of the MIS. For instance, it combines the status and the priority of a vertex in a single byte, thus reducing the memory footprint. It utilizes partially random priority values that are inversely proportional to a vertex's degree, which enables the code to find relatively large sets. ECL-MIS is 11.5× faster and produces 10% larger sets than the best prior MIS implementations for GPUs [23].

5) Minimum spanning tree: The minimum spanning tree (MST) of a weighted undirected graph is the set of edges that connect all vertices while having the least total edge weight. The ECL-MST code [14] employs a data-driven algorithm. To improve performance, it uses implicit path compression in the union-find data structure, primarily edge-based processing, and a hybrid parallelization between thread and warp computation. ECL-MST is 4.5× faster than the fastest prior GPU implementations of MST [24].

6) Strongly connected components: A strongly connected component (SCC) is a subset of vertices in a directed graph that has a path to every other vertex in the subset. The ECL-SCC code [15] is based on a new data-driven, edge-centric, parallel algorithm for computing SCCs. It identifies the maximum ID vertex on the incoming and outgoing paths of each vertex to demarcate multiple SCCs concurrently. This allows all vertices to simultaneously act as pivots, thus increasing the parallelism. It exploits the monotonicity of the maximum-ID propagation to speed up the code. On average, ECL-SCC outperforms the previous fastest parallel SCC implementations by about 2× on power-law graphs and 6.5× on mesh graphs [25].

III. Related Work

Many benchmark suites with parallel codes exist. Some include codes with known bugs, such as program verification suites. However, there is almost no prior work on the performance impact of correcting data races.

Splash-3 [26] is a parallel benchmark suite for CPUs that corrects performance bugs and data races present in the widely-used Splash-2 [27] suite, complying with the newer C11 memory consistency standards. The authors compare the runtime and lock contention of both suites on simulated hardware. Their results suggest that proper synchronization can increase performance on some programs while decreasing it on others. Our study shows that this is also the case for GPU codes.

Indigo3 [28] is a parallel program verification benchmark suite consisting of irregular graph codes with implementations for CPUs and GPUs. The suite contains 2516 bug-free codes and 39,274 buggy codes derived from 7 core graph algorithms. Indigo3 is a follow-up to the Indigo [29] benchmark suite, which consists of short parallel code patterns for CPUs and GPUs. The buggy codes in both suites are labelled for use by verification tools. Whereas Indigo and Indigo3 contain versions of the same codes with and without data races, the respective publications do not discuss the performance difference between them. Moreover, Indigo3 is based on straightforward implementations of the included graph algorithms. In contrast, we study some of the highest-performing implementations in this paper.

DataRaceBench [30] is a suite of regular programs designed to evaluate CPU data-race-detection tools. It includes a set of kernels, some of which contain data races. Verma et al. enhanced the suite by adding kernels that represent additional patterns and include FORTRAN code [31]. RMARaceBench [32] 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. Whereas these suites focus on testing tools for finding and eliminating data races, we study the performance impact of removing races.

Many other publications present parallelized and optimized irregular graph codes. Lonestar [33] contains 22 C++ and CUDA implementations of iterative graph algorithms. Pannotia [34] is an OpenCL suite of 8 applications for studying graph algorithms on GPUs. GraphBIG [35] contains implementations of representative data structures, workloads, and data sets from 21 real-world use cases of multiple application domains. GAPBS [36] 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 [37] is a suite for studying irregular graph algorithms on accelerators. It includes 9 workloads from graph analytics, sparse linear algebra, and machine learning. GBBS [38] is a C++ suite of scalable, provably-efficient implementations of 20 graph problems for multicore machines. None of these works focus on data races or their performance impact.

Boehm discusses the concept of supposedly "benign" data races and explains why they are not benign in source code [2]. Harmless races can only exist in machine code, where they are, for example, used to implement synchronization primitives. As a consequence, the standards for many high-level programming languages, such as C [3], C++ [4], CUDA [7], OpenMP [6], and Posix threads [5], define data races as undefined behavior. Hence, there is no guarantee that source code containing a data race will compile into correct machine code. This triggered our study.

Nevertheless, there are several dynamic verification tools that categorize data races in CPU code as harmful or "harmless" for triage and performance purposes. RaceFuzzer [39], Portend [40], RaceMob [41], RaceChecker [42], ColFinder [43], PRFinder [44], and RaceTest [45] employ various methods to identify potential data races and then verify each one, tagging or simply ignoring data races found to be "benign". The existence of these tools hints at a widespread use of data races in high-performance codes.

IV. Approach

Most of the ECL baseline codes contain data races. We identified them with the help of multiple verification tools, including Compute Sanitizer [46] and iGuard [47]. Building tools for identifying data races is an active research area, and even the best tools currently available have drawbacks. For example, iGuard seems to ignore the implicit barrier between kernel launches, causing false positive reports, and ComputeSanitizier does not check for races in global memory. Nevertheless, such tools are very useful for highlighting code regions that should be investigated in more detail, so we manually checked the source codes as well.

A. Races Found

The **APSP** code is the only *regular* program [1] in the suite. It does not have irregular control flow or irregular memory-access patterns since it processes all elements of a shared matrix and uses constant strides. As a consequence, only one thread accesses a given matrix element, meaning the baseline code does not have any data races.

The remaining studied codes are *irregular*, operate on a shared graph represented in CSR format [48], and provide no guarantee as to which threads might access a vertex or edge. Since these accesses are performed with unprotected loads and stores, all of those codes contain data races.

The **CC** code stores the label of each vertex in a shared *int* array. During computation, these labels can be read and written by any thread operating on the same connected component. Most of these accesses are unprotected.

The **GC** code records the possible colors and chosen color of each vertex in shared *int* arrays. During computation, vertices read and write their neighbors' values in these arrays using unprotected accesses.

The **MIS** code stores the combined status and priority value of each vertex in a shared *char* array. During computation, vertices read and update their neighbor's values in this array. All of these accesses are unprotected.

The **MST** code records the best neighbor to merge next for each union in a shared *long long* array. During

```
#include <cuda/atomic>
const auto relaxed = cuda::memory_order_relaxed;

template <typename T>
__device__ inline T atomicRead(T* const p)
{
  return ((cuda::atomic<T>*)p)->load(relaxed);
}

template <typename T>
__device__ inline void atomicWrite(T* const p,
    const T val)
{
  ((cuda::atomic<T>*)p)->store(val, relaxed);
}
```

Fig. 2: Atomic read and write operations using libcu++

(a) Baseline MIS

```
1 typedef unsigned char stat;
2 stat nv = node_stat[v];
```

(b) Atomic MIS

```
1 typedef unsigned char stat;
2 int* const nstat4 = (int*)node_stat;
3 stat nv = (atomicRead(&nstat4[v / 4]) >>
      ((v % 4) * 8)) & 0xff;
```

Fig. 3: Example of atomically reading a *char* using typecasting and masking in MIS

(a) Baseline MIS

```
typedef unsigned char stat;
node_stat[v] = 0x00;
```

(b) Atomic MIS

```
typedef unsigned char stat;
int* const nstat4 = (int*)node_stat;
atomicAnd(&nstat4[v / 4], ~(0xff <<
    ((v % 4) * 8)));
```

Fig. 4: Example of atomically writing 0x00 to a *char* using typecasting and masking in MIS

computation, these values can be read and written by any thread operating on the same component. Most of these accesses are unprotected.

The **SCC** code stores information about the incoming and outgoing paths of each vertex in a shared *int2* array. This built-in CUDA data type stores a pair of *int* variables as a structure [7]. The code also uses a global *boolean* variable to determine if another iteration is needed. During computation, these values can be read and written by any thread. All of these accesses are unprotected.

B. Atomic Reads and Writes

All studied codes except APSP contain at least one shared data structure in which threads may simultaneously access the same location. For example, when performing the label propagation for computing connected components, different threads may read and write the same vertex's label. To remove these data races, we replaced all memory accesses to shared data with atomic load and store operations from *libcu++*, as shown in Fig. 2. These operations use the relaxed memory ordering for maximum performance. The relaxed ordering is sufficient since there is no ordering constraint on these operations in the baseline codes. After all, we are only using atomic accesses to prevent word tearing and to force the compiler to not optimize the access away.

C. Typecasting and Masking

CUDA atomic operations only support certain data types such as *int* and *long*. They do not support small scalars (e.g., *char* and *bool*) or structures (e.g., *pair* and *int2*). However, the baseline codes use these smaller data types and structures to save memory and improve performance. To enable atomic operations on shared data structures of these types, we use typecasting and masking. For instance, the MIS code stores the vertex status in a *char* variable, as shown in Fig. 3a. Since the atomic load operation in CUDA does not support *char*, we use the workaround shown in Fig. 3b, which casts the type to *int* on line 2, computes the new index (i.e., $v/4$) for the *int* array, atomically reads the *int*, and then uses bit shifting and masking to extract the needed *char* in line 3.

To atomically write these unsupported types, we use a similar method based on atomic bitwise operations. For example, Fig. 4a writes 0x00 to the vertex status. Fig. 4b creates a mask and uses it with an atomic bitwise AND operation to zero out the specified *char* from the *int* array.

Since the SCC code's shared *boolean* is a single scalar variable (i.e., not an array), we simply changed its type to an *int* to enable atomic operations on it. For the *int2* variables, we change the type to *long long* for which atomic operations exist. In cases where the code accesses only one value of such a pair, we use helper functions to typecast each half of the 64-bit value to an *int* and apply atomic operations to the specified half, as shown in Fig. 5. Note that word tearing between the upper and lower halves of the *long long* value is okay in these cases but not within the halves.

V. Experimental Methodology

A. Hardware and Software

We compared the performance of our race-free CUDA codes with their baseline counterparts on 4 generations of NVIDIA GPUs. Table I lists the specifications of these GPUs, where the cores denote the processing elements in the streaming multiprocessors (SMs). Each SM has an L1 cache and all SMs share an L2 cache. The last two columns of the table list the nvcc version and flags used to compile the codes. Since nvcc 10.1 predates the introduction of libcu++,

```
static __device__ int readFirst(ull* addr)
{
  int* iaddr = (int*)addr;
  return atomicRead(&iaddr[0]);
}


static __device__ int readSecond(ull* addr)
{
  int* iaddr = (int*)addr;
  return atomicRead(&iaddr[1]);
}

static __device__ void writeFirst(ull* const
    addr, const int first)
{
  int* const iaddr = (int*)addr;
  atomicWrite(&iaddr[0], first);
}


static __device__ void writeSecond(ull* const
    addr, const int second)
{
  int* const iaddr = (int*)addr;
  atomicWrite(&iaddr[1], second);
}
```

Fig. 5: Functions for atomic read and write operations on individual *int* values stored as pairs in *long long* variables

we used the version of libcu++ from the CUDA C++ Common Libraries (CCCL) [49] for the Titan V runs.

B. Inputs

Since the baseline version of APSP does not have any data races, we do not measure its performance. We ran CC, GC, MIS, and MST on the undirected inputs shown in Table II. These inputs are available on an ECL web page [50]. For SCC, we used a mix of mesh and power-law directed graphs from the ECL-SCC paper [25], which are listed in Table III. All input graphs are stored in compressed-sparse-row (CSR) format [48] and vary significantly in size, type, degree, and origin. We run each code nine times per input and use the median runtime for comparison.

VI. Results

A. Performance of Race-free Programs

Tables IV to VII present the speedups (the baseline's runtime divided by that of the race-free program) for CC, GC, MIS, and MST on our four test GPUs. The SCC speedups are listed separately in Table VIII since they are based on different inputs. The bottom rows of each table show the minimum, geometric-mean, and maximum speedup for each algorithm. The geometric-mean speedups are visualized in Fig. 6. A speedup greater than 1 means the race-free program is faster, and less than 1 means the race-free program runs more slowly than the baseline. In the tables, speedups of 1 or greater are highlighted in green. Except on a few of the smallest inputs, the nine repeated runs of each configuration are very close in runtime to each other. The median relative deviation is only 0.6%.

The race-free CC and SCC codes are substantially slower than the baseline on all 4 devices, with the least performance loss on the 2070 Super. The race-free GC and MST codes are slightly slower than their baselines, with their mean speedups staying above 0.92. Interestingly, the race-free MIS code is 5-11% faster than its baseline on all 4 GPUs.

MIS and GC only update each vertex's direct neighbors. As a consequence, these updates typically do not overlap. In contrast, CC, MST, and SCC repeatedly access the set representative of each vertex, leading to more overlapping atomic accesses and lower performance. The impact on MST is significantly lower due to its use of implicit path compression, which reduces the number of these accesses [24]. In CC and SCC, the newly inserted atomic operations replace accesses to non-volatile arrays in the baseline codes, affecting L1 cache performance much more significantly than in the other codes. For example, the baseline CC code includes a particularly significant code section with data races. This code uses a non-volatile array of pointers to find its set representative and shortens the path along the way, called pointer jumping. These updates are monotonic, so they do not need to be visible to other threads for correctness. However, the race-free CC code performs an atomic read and an atomic write for every jump. Profiling the two code versions revealed that the baseline code has a much higher L1 hit rate for both loads and stores, which explains the performance difference.

The MIS code is based on an asynchronous implementation where threads repeatedly poll neighbors and eventually update a vertex. Since the baseline code does this using non-atomic loads and stores, the compiler may "optimize" some of these accesses, thus delaying when updates become visible to other threads. The use of atomic loads and stores in our race-free version may prevent these changes, resulting in faster propagation of values and, therefore, improved performance. Profiling the MIS code reveals increased cache hit rates, supporting this theory. In addition, the MIS source code changed the most relative to its baseline (see Figs. 3b and 4b), which yields correspondingly larger differences in the compiled machine instructions.

B. Correlation with Graph Properties

Table IX lists the correlation coefficients between the input graphs' properties and the resulting speedups. Note that the low variance in speedups exhibited by GC and MST causes outliers to be over-represented in their correlations.

The degree distribution of the input graph only significantly affects the SCC code. Its speedup shows a moderate to strong negative correlation with the average degree.

The graph size affects the race-free speedup quite differently depending on the code and GPU. It is not a significant

TABLE I: GPU specifications and compilation parameters

GPU Name	Architecture	Cores	SMs	L1 Size	L2 Size	Memory	Mem. Bandwidth	NVCC	NVCC Flags
Titan V	Volta	5120	80	96 kB	4.5 MB	12 GB	652 GB/s	10.1	-O3 -arch=sm_70
2070 Super	Turing	2560	40	96 kB	4 MB	8 GB	448 GB/s	12.0	-O3 -arch=sm_75
A100	Ampere	6912	108	192 kB	40 MB	40 GB	1555 GB/s	12.0	-O3 -arch=sm_80
4090	Ada Lovelace	16,384	128	128 kB	72 MB	24 GB	1008 GB/s	12.0	-O3 -arch=sm_89

TABLE II: Undirected input graphs for CC, GC, MIS, and MST

Graph Name	Edges	Vertices	Type	d-avg	d-max
2d-2e20.sym	4,190,208	1,048,576	grid	4.0	4
amazon0601	4,886,816	403,394	co-purchases	12.1	2,752
as-skitter	22,190,596	1,696,415	Internet topology	13.1	35,455
citationCiteseer	2,313,294	268,495	publication citations	8.6	1,318
cit-Patents	33,037,894	3,774,768	patent citations	8.8	793
coPapersDBLP	30,491,458	540,486	publication citations	56.4	3,299
delaunay_n24	100,663,202	16,777,216	triangulation	6.0	26
europe_osm	108,109,320	50,912,018	roadmap	2.1	13
in-2004	27,182,946	1,382,908	weblinks	19.7	21,869
internet	387,240	124,651	Internet topology	3.1	151
kron_g500-logn21	182,081,864	2,097,152	Kronecker	86.8	213,904
r4-2e23.sym	67,108,846	8,388,608	random	8.0	26
rmat16.sym	967,866	65,536	RMAT	14.8	569
rmat22.sym	65,660,814	4,194,304	RMAT	15.7	3,687
soc-LiveJournal1	85,702,474	4,847,571	community	17.7	20,333
USA-road-d.NY	730,100	264,346	roadmap	2.8	8
USA-road-d.USA	57,708,624	23,947,347	roadmap	2.4	9

TABLE III: Directed input graphs for SCC

Graph Name	Edges	Vertices	Type	d-avg	d-max
cage14	27,130,349	1,505,785	power-law	18.02	41
circuit5M	59,524,291	5,558,326	power-law	10.71	1,290,501
cold-flow	6,295,941	2,112,512	mesh	2.98	5
flickr	9,837,214	820,878	power-law	11.98	10,272
klein-bottle	18,793,715	8,388,608	mesh	2.24	4
star	654,080	327,680	mesh	2.00	2
toroid-hex	4,684,142	1,572,864	mesh	2.98	4
toroid-wedge	487,798	196,608	mesh	2.48	4
web-Google	5,105,039	916,428	power-law	5.57	456
wikipedia	39,383,235	3,148,440	power-law	12.51	6,576

TABLE IV: Speedups of race-free codes on Titan V

Input	CC	GC	MIS	MST
2d-2e20.sym	0.55	1.00	1.16	0.97
amazon0601	0.57	1.00	1.49	0.98
as-skitter	0.66	1.01	2.05	0.99
citationCiteseer	0.51	0.98	1.12	0.93
cit-Patents	0.80	1.00	1.21	0.96
coPapersDBLP	0.52	1.02	0.99	0.96
delaunay_n24	0.81	1.00	1.00	0.98
europe_osm	0.99	1.00	1.01	0.99
in-2004	0.59	0.97	1.11	0.99
internet	0.55	1.00	1.00	0.97
kron_g500-logn21	0.75	0.99	1.10	0.92
r4-2e23.sym	0.76	1.00	0.99	0.95
rmat16.sym	0.59	1.00	1.04	0.97
rmat22.sym	0.86	0.99	1.08	0.95
soc-LiveJournal1	0.72	1.00	0.98	0.97
USA-road-d.NY	0.47	1.00	0.91	0.99
USA-road-d.USA	0.73	1.00	1.05	0.98
Min Speedup	**0.47**	**0.97**	**0.91**	**0.92**
Geomean Speedup	**0.66**	**1.00**	**1.11**	**0.97**
Max Speedup	**0.99**	**1.02**	**2.05**	**0.99**

TABLE V: Speedups of race-free codes on 2070 Super

Input	CC	GC	MIS	MST
2d-2e20.sym	0.80	0.98	1.05	0.98
amazon0601	0.81	1.00	1.28	0.95
as-skitter	0.88	0.99	1.70	0.98
citationCiteseer	0.63	0.98	1.05	0.92
cit-Patents	0.95	1.00	1.06	0.98
coPapersDBLP	0.84	0.97	0.99	0.96
delaunay_n24	2.09	1.00	0.99	1.00
europe_osm	1.22	1.00	0.98	1.00
in-2004	0.85	0.94	0.99	0.98
internet	0.63	0.99	1.05	0.84
kron_g500-logn21	0.96	0.99	1.03	0.97
r4-2e23.sym	1.12	1.00	0.94	0.97
rmat16.sym	0.70	0.94	1.03	0.88
rmat22.sym	0.97	0.99	1.01	0.97
soc-LiveJournal1	1.00	1.00	1.00	0.98
USA-road-d.NY	0.54	0.87	0.94	0.89
USA-road-d.USA	0.73	1.00	0.98	1.00
Min Speedup	**0.54**	**0.87**	**0.94**	**0.84**
Geomean Speedup	**0.88**	**0.98**	**1.05**	**0.95**
Max Speedup	**2.09**	**1.00**	**1.70**	**1.00**

TABLE VI: Speedups of race-free codes on A100

Input	CC	GC	MIS	MST
2d-2e20.sym	1.11	1.00	1.06	0.94
amazon0601	0.86	1.00	1.42	0.96
as-skitter	0.39	1.00	1.81	0.98
citationCiteseer	1.01	0.99	1.16	0.92
cit-Patents	0.66	0.99	1.02	0.91
coPapersDBLP	0.66	0.93	0.97	0.93
delaunay_n24	0.56	1.00	0.90	1.00
europe_osm	0.36	1.00	0.90	0.99
in-2004	0.60	0.93	1.00	1.02
internet	1.18	1.00	1.16	0.86
kron_g500-logn21	0.52	0.96	1.39	0.87
r4-2e23.sym	0.39	1.00	0.95	0.91
rmat16.sym	1.43	0.99	1.05	0.89
rmat22.sym	0.59	1.00	0.97	0.89
soc-LiveJournal1	0.49	0.97	0.98	0.93
USA-road-d.NY	0.62	1.00	1.01	0.89
USA-road-d.USA	0.67	1.00	0.93	0.98
Min Speedup	**0.36**	**0.93**	**0.90**	**0.86**
Geomean Speedup	**0.66**	**0.99**	**1.08**	**0.93**
Max Speedup	**1.43**	**1.00**	**1.81**	**1.02**

TABLE VII: Speedups of race-free codes on 4090

Input	CC	GC	MIS	MST
2d-2e20.sym	0.47	0.99	1.10	0.99
amazon0601	0.43	0.99	1.34	0.99
as-skitter	0.42	0.75	1.70	0.98
citationCiteseer	0.50	1.00	1.14	0.96
cit-Patents	0.48	1.24	1.04	0.91
coPapersDBLP	0.42	0.75	0.95	0.90
delaunay_n24	0.39	1.00	0.97	0.97
europe_osm	0.41	1.00	0.98	0.99
in-2004	0.48	0.80	1.11	1.00
internet	0.48	0.99	0.97	0.99
kron_g500-logn21	0.49	0.89	1.27	0.92
r4-2e23.sym	0.31	1.00	0.98	0.92
rmat16.sym	0.69	0.99	1.10	0.98
rmat22.sym	0.46	1.00	0.90	0.91
soc-LiveJournal1	0.45	0.99	0.99	0.96
USA-road-d.NY	0.47	1.07	0.90	0.99
USA-road-d.USA	0.42	0.99	0.97	0.96
Min Speedup	**0.31**	**0.75**	**0.90**	**0.90**
Geomean Speedup	**0.45**	**0.96**	**1.07**	**0.96**
Max Speedup	**0.69**	**1.24**	**1.70**	**1.00**

TABLE VIII: Speedups of race-free SCC

Input	Titan V	2070 S	A100	4090
cage14	0.52	0.67	0.36	0.43
circuit5M	0.77	0.76	0.98	0.79
cold-flow	0.90	0.84	0.57	0.47
flickr	0.43	0.69	0.34	0.40
klein-bottle	0.58	0.85	0.36	0.49
star	0.98	0.84	0.78	1.07
toroid-hex	0.90	0.81	0.58	0.53
toroid-wedge	1.05	0.86	0.87	1.06
web-Google	0.72	0.96	0.37	0.45
wikipedia	0.83	0.85	0.27	0.30
Min Speedup	**0.43**	**0.67**	**0.27**	**0.30**
Geomean Speedup	**0.74**	**0.81**	**0.50**	**0.55**
Max Speedup	**1.05**	**0.96**	**0.98**	**1.07**

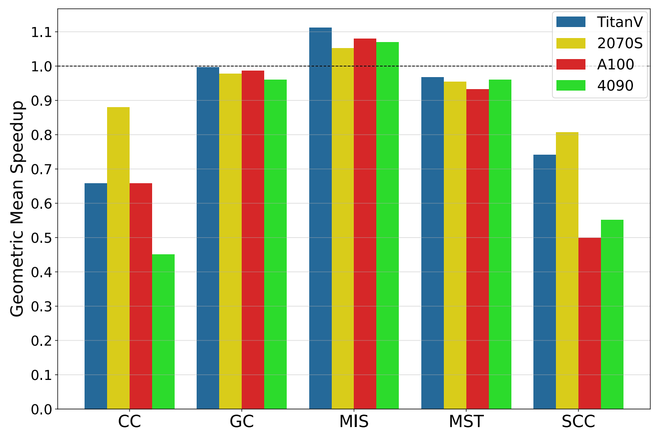

Fig. 6: Geometric-mean speedup over the baseline across all inputs on all tested GPUs

factor for the race-free GC and MST codes, which exhibit little variance in speedup between inputs. Our MIS code shows a mild but consistent decrease in speedup as the vertex count of the input increases. The speedup of CC is greatly affected by the size of the input and the GPU used. On the Titan V and 2070 Super devices, CC's speedup increases with the graph size. However, on the newer A100 and 4090 GPUs, we see the opposite, where CC has worse speedup on larger graphs.

C. *Race-free Conversion Performance Impact*

As the many green entries in Tables IV to VIII illustrate, eliminating data races does not always slow down the program and can even speed it up in some cases. The algorithm implementation, the underlying data structure, and the GPU architecture all affect the cumulative overhead of the code inserted to make the program race-free. For example, the race-free CC code is slower across most inputs and devices but over twice as fast on a specific input and GPU. Naturally, and as already mentioned, the execution frequency of the affected code section plays an important role in determining the performance impact. Overall, we observe a trend towards more slowdown on newer GPUs as illustrated in Fig. 6.

VII. Summary and Conclusions

We identify "benign" data races in five high-performance graph analytics CUDA codes and explain how these races, despite their apparent harmlessness, are problematic. Then, we modified the codes to make them data-race-free and study the performance impact of those code changes on 4 GPUs using graphs from various domains. The resulting validated and fully optimized race-free graph analytics codes are publicly available in the open-source ECL-Suite [16].

Our results show that, despite the additional synchronization, the race-free versions are not necessarily slower than

TABLE IX: Correlation coefficients between input graph properties and observed speedups

Correlated with	CC	GC	MIS	MST	SCC
	Titan V				
Edge Count	0.71	-0.07	-0.21	-0.35	-0.27
Vertex Count	0.72	0.09	-0.20	0.32	-0.30
Average Degree	-0.03	0.03	-0.04	-0.57	-0.62
	2070 Super				
Edge Count	0.57	0.41	-0.23	0.54	-0.34
Vertex Count	0.43	0.32	-0.21	0.47	0.01
Average Degree	-0.02	0.03	-0.04	0.11	-0.68
	A100				
Edge Count	-0.60	-0.16	-0.10	0.04	0.09
Vertex Count	-0.42	0.26	-0.37	0.47	-0.07
Average Degree	-0.17	-0.67	0.25	-0.28	-0.37
	4090				
Edge Count	-0.30	-0.06	-0.07	-0.38	-0.21
Vertex Count	-0.35	0.14	-0.25	0.14	-0.23
Average Degree	0.10	-0.48	0.19	-0.52	-0.47

their baseline counterparts. Whereas our modified CC and SCC codes are 1.1 to 2.2 times slower than their original implementations, the race-free MIS code is actually 5-11% faster on average. The verified GC and MST codes are less affected, running 0-8% slower on average.

Implementations like CC and SCC that rely heavily on racy non-volatile accesses lose a substantial amount of performance when those accesses are converted. In contrast, graph algorithms that already use volatile data structures do not incur much slowdown. In some asynchronous codes, the atomic operations needed to make them race-free can prevent undesired compiler optimizations that may delay when the latest information is made available to other threads, thus speeding up the program execution.

These results show that the removal of "benign" data races can have a neutral or positive effect on performance in addition to reducing undefined behavior. This further endorses the removal of data races as a best practice approach in parallel programming and teaching.

We found recent GPUs to be more negatively affected by extra synchronization than older GPUs. Hence, the performance gap between racy and non-racy code might increase in the future. We hope that hardware manufacturers recognize this trend and the importance of race-free codes and add more support for fast atomics in future GPUs.

References

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

[2] H.-J. Boehm, "How to miscompile programs with "benign" data races," in *Proceedings of the 3rd USENIX conference on Hot topic in parallelism*, ser. HotPar'11. USENIX Association, 2011, p. 3. [Online]. Available: https://www.usenix.org/legacy/events/hotpar11/tech/final_files/Boehm.pdf

[3] ISO, *ISO/IEC 9899:2011 Information technology — Programming languages — C*. Geneva, Switzerland: International Organization for Standardization, Dec. 2011. [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853

[4] ISO, *ISO/IEC 14882:2011 Information technology — Programming languages — C++*, 3rd ed. Geneva, Switzerland: International Organization for Standardization, Sep. 2011. [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372

[5] IEEE, "IEEE standard for information technology—portable operating system interface (POSIX(TM)) base specifications, issue 7," *IEEE Std 1003.1, 2016 Edition (incorporates IEEE Std 1003.1-2008, IEEE Std 1003.1-2008/Cor 1-2013, and IEEE Std 1003.1-2008/Cor 2-2016)*, pp. 1–3957, 2016.

[6] OpenMP Architecture Review Board, "OpenMP application program interface version 5.2," Nov. 2021. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

[7] NVIDIA, "CUDA C++ Programming Guide," https://docs.nvidia.com/cuda/archive/12.4.0/cuda-c-programming-guide/index.html, accessed: May 2024.

[8] NVIDIA, "libcu++," https://nvidia.github.io/libcudacxx/, accessed: May 2024.

[9] Y. Liu, N. Azami, A. Vanausdal, and M. Burtscher, "Choosing the best parallelization and implementation styles for graph analytics codes: Lessons learned from 1106 programs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3581784.3607038

[10] Y. Liu and M. Burtscher, "ECL-APSP website," https://userweb.cs.txstate.edu/~burtscher/research/ECL-APSP/, 2021, accessed: May 2024.

[11] J. Jaiganesh and M. Burtscher, "ECL-CC website," https://userweb.cs.txstate.edu/~burtscher/research/ECL-CC/, 2018, accessed: May 2024.

[12] G. Alabandi, E. Powers, and M. Burtscher, "ECL-GC website," https://userweb.cs.txstate.edu/~burtscher/research/ECL-GC/, 2020, accessed: May 2024.

[13] M. Burtscher, S. Devale, S. Azimi, J. Jaiganesh, and E. Powers, "ECL-MIS website," https://userweb.cs.txstate.edu/~burtscher/research/ECL-MIS/, 2018, accessed: May 2024.

[14] A. Fallin, A. Gonzalez, J. Seo, and M. Burtscher, "ECL-MST website," https://userweb.cs.txstate.edu/~burtscher/research/ECL-MST/, 2023, accessed: May 2024.

[15] G. Alabandi, W. Sands, G. Biros, and M. Burtscher, "ECL-SCC website," https://userweb.cs.txstate.edu/~burtscher/research/ECL-SCC/, 2023, accessed: May 2024.

[16] Y. Liu, A. VanAusdal, and M. Burtscher, "ECL-Suite Git Repository," https://github.com/burtscher/ECL-Suite, 2024, accessed: 2024-08-05.

[17] R. W. Floyd, "Algorithm 97: Shortest path," *Commun. ACM*, vol. 5, no. 6, p. 345, Jun. 1962. [Online]. Available: https://doi.org/10.1145/367766.368168

[18] P. Sao, H. Lu, R. Kannan, V. Thakkar, R. Vuduc, and T. Potok, "Scalable all-pairs shortest paths for huge graphs on multi-GPU clusters," in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 121–131. [Online]. Available: https://doi.org/10.1145/3431379.3460651

[19] J. Jaiganesh and M. Burtscher, "A high-performance connected components implementation for GPUs," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 92–104. [Online]. Available: https://doi.org/10.1145/3208040.3208041

[20] M. T. Jones and P. E. Plassmann, "A parallel graph coloring heuristic," *SIAM J. Sci. Comput.*, vol. 14, no. 3, p. 654–669, May 1993. [Online]. Available: https://doi.org/10.1137/0914041

[21] G. Alabandi, E. Powers, and M. Burtscher, "Increasing the parallelism of graph coloring via shortcutting," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 262–275. [Online]. Available: https://doi.org/10.1145/3332466.3374519

[22] M. Luby, "A simple parallel algorithm for the maximal independent set problem," *SIAM J. Comput.*, vol. 15, no. 4, p. 1036–1055, Nov. 1986. [Online]. Available: https://doi.org/10.1137/0215074

[23] M. Burtscher, S. Devale, S. Azimi, J. Jaiganesh, and E. Powers, "A high-quality and fast maximal independent set implementation for GPUs," *ACM Trans. Parallel Comput.*, vol. 5, no. 2, Dec. 2018. [Online]. Available: https://doi.org/10.1145/3291525

[24] A. Fallin, A. Gonzalez, J. Seo, and M. Burtscher, "A High-Performance MST Implementation for GPUs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3581784.3607093

[25] G. Alabandi, W. Sands, G. Biros, and M. Burtscher, "A GPU algorithm for detecting strongly connected components," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3581784.3607071

[26] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016, pp. 101–111.

[27] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *Proceedings 22nd Annual International Symposium on Computer Architecture*, 1995, pp. 24–36.

[28] Y. Liu, N. Azami, A. VanAusdal, and M. Burtscher, "Indigo3: A parallel graph analytics benchmark suite for exploring implementation styles and common bugs," *ACM Trans. Parallel Comput.*, May 2024. [Online]. Available: https://doi.org/10.1145/3665251

[29] Y. Liu, N. Azami, C. Walters, and M. Burtscher, "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, 2022, pp. 24–34.

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

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

[32] S. Schwitanski, J. Jenke, S. Klotz, and M. S. Müller, "RMARaceBench: A microbenchmark suite to evaluate race detection tools for RMA programs," in *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 205–214. [Online]. Available: https://doi.org/10.1145/3624062.3624087

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

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

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

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

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

[38] L. Dhulipala, J. Shi, T. Tseng, G. E. Blelloch, and J. Shun, "The graph based benchmark suite (GBBS)," in *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, ser. GRADES-NDA'20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3398682.3399168

[39] K. Sen, "Race directed random testing of concurrent programs," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 11–21. [Online]. Available: https://doi.org/10.1145/1375581.1375584

[40] B. Kasikci, C. Zamfir, and G. Candea, "Data races vs. data race bugs: telling the difference with portend," *SIGPLAN Not.*, vol. 47, no. 4, p. 185–198, Mar. 2012. [Online]. Available: https://doi-org.libproxy.txstate.edu/10.1145/2248487.2150997

[41] B. Kasikci, C. Zamfir, and G. Candea, "RaceMob: crowdsourced data race detection," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 406–422. [Online]. Available: https://doi.org/10.1145/2517349.2522736

[42] K. Lu, Z. Wu, X. Wang, C. Chen, and X. Zhou, "RaceChecker: Efficient identification of harmful data races," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Mar. 2015, pp. 78–85.

[43] Z. Wu, K. Lu, X. Wang, and X. Zhou, "Collaborative technique for concurrency bug detection," *International Journal of Parallel Programming*, vol. 43, no. 2, pp. 260–285, Apr. 2015. [Online]. Available: https://doi.org/10.1007/s10766-014-0304-y

[44] Z. Wu, K. Lu, X. Wang, X. Zhou, and C. Chen, "Detecting harmful data races through parallel verification," *The Journal of Supercomputing*, vol. 71, no. 8, pp. 2922–2943, Aug. 2015. [Online]. Available: https://doi.org/10.1007/s11227-015-1418-8

[45] Z. Ding and Z. Zhou, "RaceTest: harmful data race detection based on testing technology in WS-BPEL," *Service Oriented Computing and Applications*, vol. 13, no. 2, pp. 141–154, Jun. 2019. [Online]. Available: https://doi.org/10.1007/s11761-019-00261-1

[46] NVIDIA, "Compute sanitizer," https://developer.nvidia.com/compute-sanitizer, accessed: May 2024.

[47] A. K. Kamath and A. Basu, "iguard: In-gpu advanced race detection," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 49–65. [Online]. Available: https://doi.org/10.1145/3477132.3483545

[48] A. George, J. W. Liu *et al.*, *Computer solution of large sparse positive definite systems.* Prentice-Hall Englewood Cliffs, NJ, 1981, vol. 134.

[49] CCCL Development Team, *CCCL: CUDA C++ Core Libraries*, 2023. [Online]. Available: https://github.com/NVIDIA/cccl

[50] M. Burtscher, "ECL graphs," https://cs.txstate.edu/~burtscher/research/ECLgraph/index.html, 2019, accessed: 2024-06-01.

Appendix

A. Abstract

Some of the fastest CUDA codes contain “benign” data races to boost their performance. However, such races can lead to unpredictable behavior and incorrect results on other hardware and compilers, making their elimination crucial for producing reliable and portable programs. This paper investigates the performance impact of removing data races from five high-end graph analytics codes. We identify and eliminate the races from these GPU programs by adding necessary synchronization and validating their correctness. We present our race-free codes and their original versions as an open-source suite. Comparing the performance of our new codes with their baseline counterparts on multiple inputs and GPUs, we observe that race-free implementations do not always incur a performance penalty. In fact, some race-free versions are faster, with our validated maximal independent set implementation achieving a 5-11% speedup. Our findings indicate that race-free code can reach comparable or even superior performance, supporting the adoption of best practices for parallel programming.

B. Artifact check-list (meta-information)

- **Algorithm:** APSP, CC, GC, MIS, MST, and SCC graph analytics algorithms
- **Compilation:** GNU C++ (`g++`) compiler and NVIDIA CUDA (`nvcc`) compilers
- **Data set:** Undirected and directed graphs
- **Run-time environment:** Results produced in a bash shell on a Linux server
- **Hardware:** Our results are from the following GPUs, but the experiment can be conducted on any NVIDIA GPU with a compute capability of at least 6.0 and should yield similar trends.
  - NVIDIA Titan V
  - NVIDIA GeForce 2070 SUPER
  - NVIDIA A100 40GB
  - NVIDIA GeForce RTX 4090
- **Metrics:** Speedup
- **Output:** Raw runtimes of the baseline and race-free codes, tables of the speedups from baseline to race-free for each algorithm on each input, and a bar chart of the geometric mean speedup of each algorithm.
- **Experiments:** Compare runtime of race-free codes against their baseline counterparts.
- **Disk space required:** At least 10 GB
- **Time needed to complete experiments:** 2-3 hours
- **Publicly available:** The code, inputs, and our results are publicly available.
- **Code license:** 3-Clause BSD
- **Data license:** 3-Clause BSD
- **Archived:** DOI: https://doi.org/10.5281/zenodo.13228335

C. Description

1) How to access: The software can be obtained from GitHub: https://github.com/burtscher/ECL-Suite.

```
git clone https://github.com/burtscher/ECL-Suite.git
```

2) Hardware dependencies: The experiment can be executed on any system that has a CUDA-enabled NVIDIA GPU with a minimum compute capability of 6.0. Compute capability values can be found here: https://developer.nvidia.com/cuda-gpus.

3) Software dependencies: All code is intended to run in a Linux environment. The requirements for reproducing the experiment are:

- `nvcc` version 10.2 or higher
- `g++` version 7.3 or higher

We provide Python scripts to automate compiling and running each code on each input multiple times as well as generating the speedup tables and figures. The requirements to run these scripts are:

- Python 3.11
- Numpy (https://numpy.org/) version 1.25 or later
- Matplotlib (https://matplotlib.org/) version 3.8 or later
- Scipy (https://scipy.org/) version 1.7 or later

4) Data sets: The inputs used for the experiment can be acquired by running the `download_inputs.sh` script.

D. Installation

Install the necessary Python packages, such as with `pip`:

```
pip3 install numpy matplotlib scipy
```

1) GPU selection: If the system has multiple GPUs, the experiment scripts use the fastest GPU available by default. If a different GPU is desired, edit the value in line 4 of `all_tests.sh`.

E. Experiment workflow

1) Acquire input graphs: To download and prepare all input graphs, run:

```
./download_inputs.sh
```

The script will create the `inputs-undirected/` and `inputs-directed/` directories and place the input graphs in them.

2) Run experiment: To compile and execute all codes on all inputs, run:

```
./all_tests.sh
```

The script will run every baseline and race-free code on every appropriate input 9 times by default, then calculate the speedups from baseline to race-free. This takes 1-2 hours on a 2070 Super.

F. Evaluation and expected results

The raw runtime logs are available in the `./results/` directory. The speedups calculated using those logs are available in the `./output/` directory. Inside, there will be two CSV files listing the speedups from the baseline to the race-free codes for each input for each algorithm (`undirected_speedups.csv` and `directed_speedups.csv`) and a figure comparing the geometric mean speedup of each algorithm (`geometric_means_bar.svg`). These are single-GPU analogues of Tables 7 and 8 and Figure 6.

The results may vary from those presented in the paper, particularly if run with different GPUs. Nonetheless, we expect the same general trends to be evident on similar hardware.

G. Experiment customization

As mentioned in 4.1, if there are multiple GPUs present, the GPU used for the experiment can be changed by editing the value of `CUDA_VISIBLE_DEVICES` on line 4 of the `all_tests.sh` script.

By default, each code is run on each input 9 times, the same number of times we used to generate the paper's results. The median of those 9 runtimes is used for the speedup calculations. If time is a concern, this number can be edited on line 5 of the `all_tests.sh` script.

H. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html