Compiler-Aided Correctness Checking of CUDA-Aware MPI Applications

Alexander Hück*, Tim Ziegler*, Simon Schwitanski†, Joachim Jenke† and Christian Bischof*
*Technical University Darmstadt, Darmstadt, Germany
{alexander.hueck, christian.bischof}@tu-darmstadt.de
tim.ziegler@stud.tu-darmstadt.de
†RWTH Aachen University, Aachen, Germany
{schwitanski, jenke}@itc.rwth-aachen.de

***Abstract*—Hybrid MPI + X models, combining the Message Passing Interface (MPI) with node-level parallel programming models, increase complexity and introduce additional correctness issues. This work addresses the challenges of detecting data races in hybrid CUDA-aware MPI applications due to the asynchronous and non-blocking nature of CUDA and MPI APIs. We introduce CuSan, an LLVM compiler extension, and runtime that tracks CUDA-specific concurrency, synchronization, and memory access semantics. We integrate CuSan with MUST, a dynamic MPI correctness tool, and ThreadSanitizer (TSan), a thread-level data race detector. MUST with TSan can already detect concurrency issues for multi-threaded MPI codes. Together with CuSan, these tools allow for comprehensive correctness checking of concurrency issues in CUDA-aware MPI applications. Our evaluation of two mini-apps reveals runtime overhead of CuSan ranging from 6× to 36×, depending on the amount of memory tracked by TSan, compared to the uninstrumented version. Memory overhead consistently remains under 1.8×. CuSan is available at https://github.com/tudasc/cusan.**

***Index Terms*—MPI, Correctness, CUDA, Data Race, ThreadSanitizer, LLVM**

I. Introduction

For efficiency, the Message Passing Interface (MPI, [1]) is often combined with node-level parallel programming models [2], [3]. This is commonly referred to as the hybrid *MPI + X* model, where X typically stands for OpenMP but can also refer to, e.g., CUDA [4].

However, MPI defines a low-level interface that is error-prone. Some dormant MPI bugs have only been uncovered after years [5], [6]. The hybrid model further increases code complexity. The combination of two programming paradigms poses correctness issues stemming from MPI [7], [8] *and* from, e.g., OpenMP [9], [10], resulting in a new set of issues [11]. Data races, such as those caused by OpenMP threads accessing memory during MPI communication, occur when these concurrent executors perform conflicting memory accesses (at least one being a write) without proper synchronization. To effectively detect such data races, MPI correctness tools [5], [6], [12]–[14] must have a holistic understanding of the program's behavior, observing all memory accesses, synchronization mechanisms, and concurrency semantics across all levels of parallelism. Tools that only observe a subset, such as by only intercepting MPI calls during runtime, will find some issues but not all [11], [15].

In this work, we present CuSan, a tool designed to address the inherent challenges of data race detection in hybrid CUDA-aware MPI applications [16]. While CUDA-aware MPI libraries [17], [18] streamline communication by allowing direct use of device pointers (eliminating the need for explicit host-device memory transfers), they introduce complexities in ensuring correct synchronization. The asynchronous APIs of both CUDA and non-blocking MPI communication places the burden on users to manually manage data dependencies and synchronize operations [19], a process that is error-prone and can lead to data races. CuSan builds upon the capabilities of two existing correctness tools, namely (i) ThreadSanitizer (TSan, [20], [21]), a dynamic data race detector for shared-memory programs, and (ii) MUST [13], a dynamic MPI correctness checker, which was already integrated with TSan [22]. We extend TSan to understand the synchronization semantics of CUDA operations by using TSan's annotation API, to ultimately facilitate the data race analysis in the context of the combined semantics of CUDA and MPI.

CuSan instruments CUDA-related code to track relevant memory accesses and synchronization events, exposing this information to TSan. MUST provides additional information about MPI (non-blocking) operations, allowing TSan to reason about the overall synchronization state of the program and detect unsynchronized access to device memory. This approach enables the detection of data races that would be missed by tools focused only on MPI or CUDA. Additionally, CuSan depends on TypeART, another tool used by MUST, to track memory allocations and their associated datatypes [23]–[25]. We extend TypeART to monitor CUDA-related memory allocations, providing CuSan with the necessary data length information to expose device memory access patterns to TSan. In summary, we make the following key contributions:

- Development of CuSan, an LLVM compiler extension and runtime, for analyzing and instrumenting CUDA codes to track CUDA domain-specific memory accesses and synchronization semantics.
- Extension of TypeART to track device memory allocations to facilitate analysis of memory access semantics.
- Integration of CuSan with MUST, TypeART and TSan, to combine MPI semantics for data race analysis of CUDA-

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00032

aware MPI applications.

The rest of the paper is structured as follows. Section II discusses the tools required to implement our CUDA-aware MPI sanitizer. In particular, we discuss (i) TSan, (ii) MUST and its integration with TSan for detecting concurrency issues in MPI, and (iii) TypeART, a tool for tracking memory allocations for MPI datatype safety. Section III introduces the CUDA-aware MPI hybrid model and discusses its synchronization requirements. Section IV covers CuSan's implementation. In Section V, we evaluate our tool with two mini-apps using CUDA-aware MPI. Section VI discusses the results and presents future work. Section VII concludes our work.

II. Background

This section gives a brief overview of the required tooling to build a CUDA-aware MPI sanitizer that integrates with the MPI correctness checker MUST.

TSan and the enabling of its API for data race detection w.r.t. CUDA are discussed in Section II-A. MUST and its ThreadSanitizer integration is discussed in Section II-B. Finally, the TypeART tool for tracking type information of memory allocations is discussed in Section II-C.

A. ThreadSanitizer

ThreadSanitizer (TSan, [20]) is packaged with Clang [21] and consists of a compiler pass and an analysis runtime library for thread-level data race analysis. The compiler pass adds calls to the runtime library to track memory accesses and function entry/exit (for stack traces). The runtime library uses shadow memory to track memory access patterns and detect potential data races. Using function interception for various pthread and C++ threading functions, the runtime tracks thread synchronization. This allows TSan to establish happens-before relationships for data race analysis. A data race occurs when two conflicting operations on the same memory location, such as a read and a write, are executed concurrently without any synchronization. A happens-before relationship between such two operations ensures that one operation is guaranteed to complete before the other begins. In that context, for any new memory access, TSan checks the shadow memory for potentially conflicting memory accesses. The happens-before relation with the previous memory access is determined by comparing the logical time of the memory access with the logical time recorded for the last synchronization with the accessing thread.

TSan provides an annotation API for users to manually expose unknown synchronization or access semantics to keep the internal bookkeeping accurate. Any synchronization can be annotated as a pair consisting of a signal (`AnnotateHappensBefore`) and a wait (`AnnotateHappensAfter`), respectively also termed release and acquire. To build the connection, the annotation functions accept a memory address as a key argument. TSan uses this key to identify the synchronization clock to store or load the vector of logical clocks. Likewise, memory accesses can be manually annotated with, e.g., `tsan_write_range`, passing a pointer and its access length (analogous for read operations).

Finally, TSan provides the abstraction of *fibers* to model user-defined concurrency [26]. MUST [22] and Archer [27] have adopted this abstraction to model the concurrency of OpenMP tasks and non-blocking MPI communication. Fibers can be instantiated by the user. Using TSan API calls, the fiber context of the executing OS thread can then be explicitly switched during the execution, modelling the concurrency of non-blocking MPI communication with MPI-specific fibers. Such fiber switches do not imply a synchronization.

B. MUST

For detecting MPI-related concurrency issues, MUST [13] uses TSan to observe memory operations of the application and combines them with the semantics of MPI calls.

a) ThreadSanitizer Integration: MPI communication presents a challenge for TSan due to two key factors: (i) Developers link against pre-compiled MPI libraries, making it impossible to instrument them with TSan's compiler pass. (ii) In typical HPC environments, network adapters can directly access communication buffers via DMA (Direct Memory Access), bypassing the normal load/store instructions that TSan would usually monitor.

To overcome these limitations, MUST takes the following approach. While intercepting an MPI communication call, MUST annotates the respective memory access semantics of MPI within TSan. For multi-threaded programs, these annotations are sufficient for TSan to detect data races between MPI calls or between MPI calls and local memory accesses by different threads.

b) Non-Blocking MPI Communication: Non-blocking MPI communication is typically used to overlap communication with computation for better overall performance. In these cases, data races might occur even within the same thread. MUST handles such races by creating a TSan fiber per non-blocking MPI communication call, annotating the buffer accesses to this fiber and synchronizing the fiber during the completion call, see Fig. 1

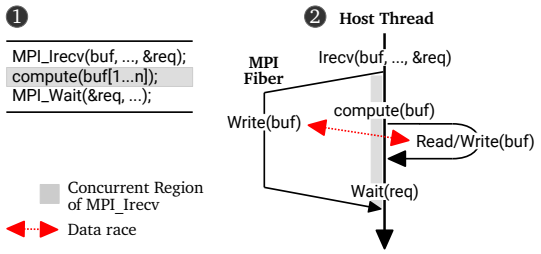

Fig. 1. (1) Non-blocking MPI calls can create data races if the user accesses a buffer between the initiation (Irecv) and completion (Wait) of a communication. (2) MUST models the concurrency using TSan fibers, enabling detection of data races within these inherently concurrent region.

C. TypeART

To detect MPI datatype-related issues, TypeART [23]–[25], an LLVM compiler extension, allows MUST to query type information of the type-less `void*` buffers passed to MPI calls.

MUST's TypeART integration is illustrated in Fig. 2. Consider the MPI function `MPI_Send(const void* buf, int count, MPI_Datatype type, ...)`. Without TypeART, MUST can neither check if the buffer arguments memory layout is compatible with that of the declared MPI datatype nor check if the count argument exceeds the allocation size.

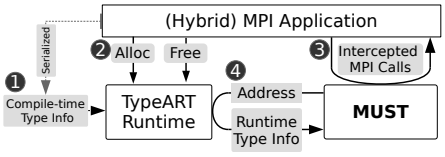

Fig. 2. TypeART as an extension to MUST, adapted from [24]. During compilation, TypeART instruments memory allocations and extracts their type information (1). A runtime tracks these allocation-related events and stores information (type, runtime allocation size) in a lookup table (2). For every intercepted MPI call (3), MUST queries the address of the type-less buffer using TypeART's runtime (4). The resulting allocation information is compared to the MPI datatype passed to the MPI call.

Compiler Extension: The compiler pass statically collects MPI-relevant memory allocations (heap, stack and globals) in the target code's LLVM intermediate representation (IR). Subsequently, our pass instruments these allocations and serializes the allocated type layouts. Each callback's arguments are (i) the allocated memory address, (ii) the runtime allocation extent, and (iii) a generated unique type id, identifying the type (layout) of the allocation. Memory de-allocation operations are tracked to keep the allocation metadata consistent with the program state.

III. CUDA-aware MPI

We introduce the relevant CUDA semantics for this work and discuss how these apply to CUDA-aware MPI.

The CUDA API allows for asynchronous operations, where kernel launches return control immediately to the host while execution happens on the device. Concurrency is expressed using CUDA streams, each enqueuing a sequence of operations (FIFO order), such as kernels, see Section III-A. Synchronization is required to ensure completion of CUDA operations w.r.t. the host. This can be achieved through explicit synchronization calls, see Section III-B1, or implicit behavior (such as device memory transfer), see Section III-B2.

Currently, CUDA-aware MPI libraries [17], [18] are not integrated with these CUDA stream semantics. Hence, these libraries only facilitate communication of device pointers without the need for host-related memory transfers. To express a data dependence between a stream and an MPI call, hence, requires explicit synchronization by the user. This applies to both non-blocking MPI or asynchronous CUDA calls. We discuss these issues in Section III-D.

A. CUDA Stream Concurrency Semantics

Streams allow for concurrent execution of device operations to maximize GPU utilization. They are created and managed by the user. By default, there is no automatic synchronization between different streams. A special default stream always exists. It has specific synchronization behavior when used alongside user-defined streams, see [28, Section 3] [29]. Mixing the default stream with user-defined streams creates logical barriers that impact the order of kernel execution, see Fig. 3: (i) Kernels on the default stream block subsequent kernels on other streams. (ii) Kernels on other streams must complete before kernels on the default stream begin. This behavior can be circumvented by explicitly marking streams *non-blocking* on creation time, or exclusively using user-defined streams in a code. If not otherwise specified, the default stream is used for kernels.

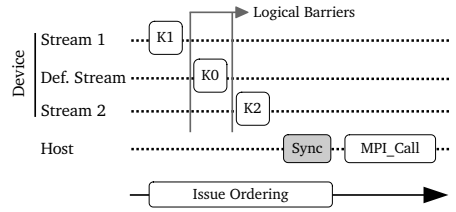

Fig. 3. Default stream semantic example. K0 waits on K1 completion, and K2 waits on K0 completion. If a user synchronizes on a stream before an MPI call, we must consider legacy stream semantics and the particular synchronization target. For instance, after a host synchronization on K2, K1 and K0 also completed.

B. CUDA Synchronization Semantics

Synchronization w.r.t. the host can be ensured with explicit synchronization API calls or implicit synchronization points, such as memory transfers. Whether the latter is synchronous w.r.t. the host depends on several factors, such as where the memory resides and which direction the memory is moved to.

1) Explicit Synchronization Calls: Explicit synchronization routines work on different granularities:

- `cudaDeviceSynchronize`: Blocks host until all streams completed on a CUDA device.
- `cudaStreamSynchronize`: Blocks host until all commands on a stream completed.
- `cudaEventSynchronize`: Blocks host until the specified event on a stream occured. CUDA events are markers on a stream that can capture completion of certain events on that stream. They are explicitly placed by the user at a specific point on a target stream and allow fine-grained control over execution ordering.

Additionally, `cudaStreamWaitEvent` is used to synchronize between streams based on some event. Further, `cudaStreamQuery` can be used to query the completion status of a stream. Hence, this call can potentially be used as a blocking 'busy-wait', looping until it succeeds. Hence, this call must be considered for synchronization.

2) Implicit Synchronization Calls: Some of the CUDA API calls have implicit synchronization behavior, particularly memory operations:

- `cudaMemcpy`: Copies data between host and device. *Generally* synchronous with respect to the host. An *async* variant exists that is *generally* asynchronous w.r.t. host.
- `cudaMemset`: Sets memory range to a specific value. An *async* variant exists. However, both are *generally* asynchronous w.r.t. host.

If the CUDA documentation states *may be* (a)synchronous, we typically interpret this pessimistically in our implementation for the purpose of data race detection.

Calls on the default stream may also be synchronous and can block the host until preceding operations (e.g., preceding kernel launches) complete, similar to Fig. 3. Likewise, memory management calls like `cudaFree` synchronize with the host across all streams [30, Appendix F] (async versions exist).

In general, functions denoted with "async" are asynchronous w.r.t. the host, while non-async CUDA functions can be considered synchronous. However, the exact behavior depends on factors like the memory type and transfer direction (as detailed in the CUDA documentation, see [28, Section 2]). We track these aspects for our data race analysis.

C. CUDA Memory Semantics

The type of memory allocation influences (implicit) synchronization behavior, see Section III-B2. For instance, pageable memory (e.g., `malloc`) and pinned (page-locked) memory (e.g., `cudaHostAlloc`) exhibit different synchronization characteristics with `cudaMemset`: The latter synchronizes with the host, while the former does not. Hence, we need to track the allocation kind to accurately reason about implicit synchronization points in CUDA operations. Additionally, CUDA-managed memory (`cudaMallocManaged`) requires explicit synchronization. While the CUDA driver automatically migrates managed memory between the host and device as necessary, operations on this memory must be synchronized to ensure a consistent memory view. In summary, the implicit synchronization semantics are complex and even depend on the memory type used.

D. CUDA-aware MPI Hybrid Model

CUDA-aware MPI libraries can eliminate the need for extra copy operations by the user, as they are able to directly access device memory pointers. To that end, they internally rely on the CUDA-specific unified virtual addressing (UVA, [31, Section 6.14]) design. UVA encodes information about the pointer memory location, allowing to differentiate between host and device memory with the CUDA API call `cuPointerGetAttribute`. Internally, these libraries can then execute device-specific communication implementations.

The UVA-design keeps the MPI interface unchanged, as needed information is passed with the buffer pointer. However, users cannot expose further CUDA semantics such as streams to the library. While CUDA expresses data dependences using the stream concept, MPI represents a gap that must be filled by the user through explicit synchronization *iff* there exists a data dependence between MPI and a stream computation. Hence, as shown in Fig. 4, we must consider two cases: (i) GPU operations followed by dependent MPI calls must explicitly synchronize, and, (ii) likewise, with a non-blocking MPI call followed by a dependent GPU operation, MPI semantics require an explicit wait before proceeding.

```
1  cudaMalloc(&d_data, size * sizeof(int));
2  if (world_rank == 0) {
3      kernel<<<...>>>(d_data, size);
4      cudaDeviceSynchronize();  // Blocks until kernel completes
5      MPI_Send(d_data, size, MPI_INT, 1, ...); // Send device data
6  } else if (world_rank == 1) {
7      MPI_Irecv(d_data, ..., &request); // Recv device data
8      MPI_Wait(&request, ...); // Blocks until Irecv completes
9      kernel_2<<<...>>>(d_data, size);
10 }
```

Fig. 4. CUDA-aware MPI example with explicit synchronization. Without synchronization in line 4, the kernel may still run on device while the subsequent send operation is executed on the host. Likewise, the wait in line 8 ensures the non-blocking Irecv completes before the kernel invocation.

Several proposals aim to bridge the gap between MPI and CUDA. Some extend MPI with explicit stream awareness [32], [33], while others introduce new APIs built on MPI as a communication layer [34], [35]. Other approaches explore alternatives to the UVA-design [19], [36], but we focus on the standard CUDA-aware MPI libraries for our work.

IV. Implementation of a CUDA-aware Sanitizer

We model the aforementioned CUDA concurrency characteristics by intercepting relevant CUDA API calls in a target code and exposing them to TSan with our CuSan tool. The interaction of CuSan, MUST, TypeART and TSan when checking CUDA-aware MPI applications is shown in Fig. 5. We compile a CUDA-aware MPI application with Clang/LLVM, which is able to compile CUDA codes [37] similar to NVIDIA's nvcc, additionally invoking CuSan's compiler pass. This process also inserts TSan instrumentation in the user code. The application is executed with MUST, which intercepts the MPI calls during runtime. CuSan and MUST call TSan's API with the relevant concurrency semantics of

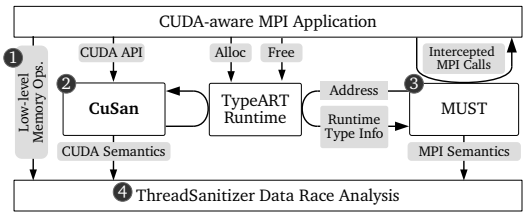

Fig. 5. CuSan with MUST. The MPI application is instrumented with our extensions (and TSan). 1) TSan tracks user host code memory accesses relevant for CUDA managed memory or MPI buffer accesses. 2) CuSan provides TSan with the particular CUDA semantics for the data race analysis. 3) Likewise, MUST provides TSan with MPI semantics. MUST and CuSan use TypeART for datatype analysis and CUDA allocation size queries, respectively. 4) TSan combines this information for data race analysis.

CUDA and MPI, respectively. Thus, TSan is able to detect data races if any are present.

We base our analysis work on TSan's *fiber* implementation, as described in Section II-A. Section IV-A discusses how we use fibers to represent the concurrent execution of a CUDA stream, how we annotate CUDA calls with happens-before synchronization and memory-related annotations. Section IV-B discusses our compiler extension which adds the required callbacks for our runtime analysis. Our implementation is based on Clang 14 and CUDA 11.5.

A. CuSan Runtime Race Detection

Our runtime integrates with TSan by using CUDA API callbacks introduced through our instrumentation (see Section IV-B). This enables us to precisely map CUDA-related events to TSan's concurrency model.

Within this model, each CUDA stream is represented as a distinct TSan fiber, mirroring the independent execution context of the device relative to the host code. Similarly, MUST extends TSan's concurrency model to encompass MPI communication. Non-blocking MPI operations are modeled as TSan fibers, capturing their asynchronous nature w.r.t. the host, see Section II-B. For blocking MPI calls, it is sufficient for MUST to annotate the memory access on the main TSan host thread.

Together, these extensions allow TSan to monitor the synchronization state of both MPI and CUDA operations, enabling CUDA-aware MPI data race detection. For instance, to prevent data races, each CUDA stream that writes to memory involved in MPI communication must be synchronized before the dependent MPI call. If a CUDA stream accesses memory without explicit synchronization using TSan's happens-before annotations, and an MPI call accesses the same location, a data race is detected if at least one operation is a write. Fig. 6 illustrates examples of both MPI-to-CUDA and CUDA-to-MPI data race scenarios modeled using this approach. Many CUDA stream fibers and MPI fibers may exist in highly concurrent CUDA-aware MPI applications.

a) Tracking Streams (and Events) with TSan fibers: Our runtime instantiates a context per CUDA device, containing (i) a lookup table for each stream to its fiber, (ii) a lookup table for CUDA events to its stream, (iii) a lookup for memory creation attributes (see Section III-C), (iv) and reference to the host (CPU) fiber. The default stream is always tracked, user-defined streams are tracked on demand at creation time. We track if a stream was created with a *non-blocking* attribute.

b) Kernel calls: We highlight the necessary steps of the data race analysis during an intercepted kernel call. To that end, we require (i) kernel argument references, (ii) their memory access mode (read/write), (iii) and the stream the kernel is run on. We execute the following TSan-related calls in order, (i) switch to the fiber of the kernel stream, (ii) for all kernel argument memory regions, mark these as read/write based on our static code analysis and TypeART for querying the dynamic extent (e.g., `tsan_write_range`), (iii) starts

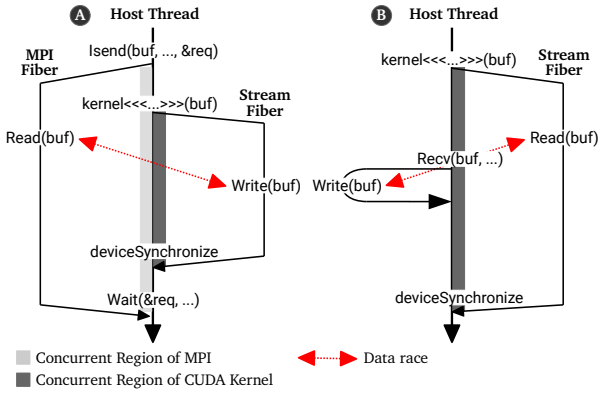

Fig. 6. Fiber interaction between MPI and CUDA kernel invocations. **A**: An `MPI_Isend` call is intercepted by MUST, which uses its internal TSan fiber reference to mark a read operation on `buf`. If a kernel is invoked before `MPI_Wait`, CuSan switches to its fiber reference for that stream to mark a write on `buf`, leading TSan to detect a data race. **B**: Similar data race scenario during a kernel execution with a blocking `MPI_Recv` which does not necessitate an extra MPI fiber. Other interleavings are possible.

a happens-before arc (`AnnotateHappenBefore`), (iv) and, finally, switch back to the CPU fiber.

c) Explicit Synchronization: Synchronization terminates a happens-before arc (`AnnotateHappensAfter`) and is executed for each supported explicit and implicit CUDA synchronization call, see Section III-B. For a specific stream or event-related synchronization, we terminate on that stream. For a `cudaDeviceSynchronize` call, we iterate over all existing streams to signal synchronization on each.

d) Implicit Synchronization: CUDA's memory operations access memory on some stream and may also be a synchronous operation w.r.t. host. Hence, we must mark the respective memory region's access mode (like with kernel calls) and optionally synchronize on the stream/device. For details on when they may synchronize, see Section III-B2.

e) Legacy Default Stream Synchronization: Special care has to be taken when user-defined streams and the default stream is used in a target code, see Section III-A. When a program synchronizes on the default stream, we terminate the happens-before arc on all *blocking* streams as they must finish before the synchronization call returns to host. Likewise, synchronization on a user-defined stream may terminate the arc of other streams, as described in Fig. 3. Synchronization on *non-blocking* streams does not exhibit this behavior.

f) Managed memory allocations: With managed memory, a user may access such allocations in user code outside of CUDA and MPI calls. TSan's compiler pass already instruments these appropriately for its analysis.

B. CuSan Compiler Pass

Clang's CUDA compilation is split into device code and host code compilation, respectively. Host code relies on the results of the device code compilation, see Fig. 7.

Device code generation is additionally running our analysis w.r.t. kernel memory accesses to determine if a kernel launch

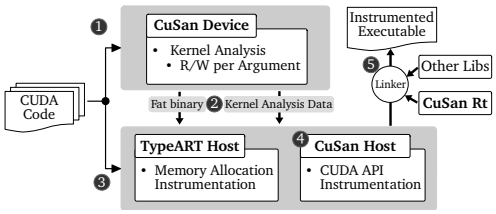

Fig. 7. Simplified view of the Clang/LLVM CUDA compilation process, for details see [37]. 1) Clang first compiles the device-specific code. 2) While the device code is compiled to a fat binary, we analyze each kernel w.r.t. read/write memory access semantics of the arguments. 3) Subsequently, the host code is compiled. 4) We instrument (CUDA) memory allocations with TypeART and, with CuSan, relevant CUDA API calls. For kernel calls we use the device code-collected memory access data. 5) Finally, the executable is linked against our runtime for the added callbacks.

reads or writes to a device pointer, see Section IV-B1. We use this analysis data during the host code compilation, adding instrumentation before, e.g., a kernel launch call, to give our runtime the necessary memory access data, see Section IV-B2.

1) Analyzing Kernel Memory Accesses: To mark the access of device memory regions, we need to analyze the particular access mode of each kernel argument. To that end, we apply a conservative interprocedural forward-dataflow analysis of the arguments of a kernel, as nested kernel calls may exist. For each argument, we return a read, write, or read/write attribute, see Fig. 8.

```
1 void kernel_nested(float* y, float* x, int tid) { y[tid] = x[tid]; }
2 void kernel(float* d_a, float* d_b) {
3     int tid = threadIdx.x + blockIdx.x * blockDim.x;
4     kernel_nested(d_a, d_b, tid);
5 }
```

Fig. 8. Example of two relevant cases: (i) The analysis follows the device pointer for `d_a` along its data flow, and reaches the function call to `kernel_nested`. The pointer is passed as the first argument, as such the analysis continues with the data flow of the first parameter `y` of the callee `kernel_nested` and detects a write operation. (ii) For `d_b`, the aliasing pointer `x`, on the other hand, is only read. In summary, `d_a` and `y` are marked `write`, whereas `d_b` and `x` are marked `read` for each kernel, respectively.

2) Instrumentation of the CUDA API: The instrumentation of the relevant CUDA API is straightforward. For each supported CUDA call, we add a callback to our runtime in the LLVM IR before the CUDA call. As arguments, we pass relevant information such as (i) kernel arguments and their memory access attributes, (ii) the stream arguments for stream-based concurrency tracking, (iii) event IDs for event-based synchronization, or (iv) memory movement attributes relevant for synchronization behavior, depending on the specific API call. We exemplify instrumentation of a CUDA kernel call in Fig. 9.

C. Extending TypeART

TypeART's extension for CUDA is straightforward, (i) we extend the compiler extension to handle the separated compilation of device and host code, and (ii) we instrument all memory allocations related to CUDA, such as `cudaMalloc`

```
1 // Device:
2 void kernel(float* d_a, float* d_b) { ... }
3 // Host:
4 void device_stub_kernel(float* d_a, float* d_b) {
5     ...
6     // 1. list of arguments, 2. list of memory attribs, 3. stream:
7     _cusan_kernel_register({d_a, d_b}, {w, r}, &a_stream);
8     cudaLaunchKernel(..., {d_a, d_b}, ..., a_stream);
9 }
```

Fig. 9. Pseudo code of a CUDA kernel call and our instrumentation. A host-side kernel invocation causes the creation of a kernel stub function in LLVM that assembles the required arguments for the call to `cudaLaunchKernel` (line 8), see [28, Section 6.7]. We take relevant arguments, and look up the read/write argument attributes for the called kernel, and pass it to our callback (line 7).

or `cudaFree`. During runtime, MUST and CuSan query type information and allocation sizes, respectively. Based on UVA, we can differentiate between host and device memory pointers.

V. Evaluation

Our evaluation focuses on performance overheads of (i) TSan, (ii) MUST, (iii) CuSan and (iv) the combination thereof compared to the unmodified application. The latter is henceforth called *vanilla*. CuSan and MUST are always executed with TSan enabled. Only CuSan uses TypeART, required for querying device pointer allocation sizes. MUST is configured to only check for data races of (non-blocking) MPI communication. Our benchmark framework is available at https://github.com/tudasc/cusan-tests.

Benchmark setup: The benchmarks were run on two compute nodes of the Lichtenberg HPC cluster of TU Darmstadt, using a NVIDIA Tesla V100 for each MPI process. We use the Clang compiler 14, OpenMPI 4.1.6 and CUDA version 11.8 with `arch=sm_70`. Default optimization flags (-O2/-O3) are used with added debug information for MUST's diagnostics. Benchmark values are the average for one process over 4 runs (with an additional warmup run that is not counted).

Benchmark applications: We evaluate our implementation based on two mini-apps, namely a C Jacobi Solver [38] and TeaLeaf [39], a C++ heat conduction solver. Both solvers work on a discretized domain and exchange boundary values with CUDA-aware MPI calls. Jacobi uses blocking MPI send-recv operations whereas TeaLeaf uses non-blocking calls. Modifications were made to the build system, as nvcc uses different CUDA-specific flags than Clang.

A. Runtime and Memory Overheads

1) Runtime: Fig. 10 shows the induced runtime overhead of the correctness tools. For Jacobi, TSan and MUST add an acceptable overhead of about 2× to 5× compared to vanilla. CuSan, however, adds a factor of 36× overhead. This is due to the large domain size which has to be tracked by TSan with each kernel invocation. This tracking is expensive, see Section V-B for more details. The combination of MUST & CuSan adds additional overhead.

Tealeaf's model, on the other hand, is a smaller domain and, hence, exhibits overall lower overheads. TSan's overhead

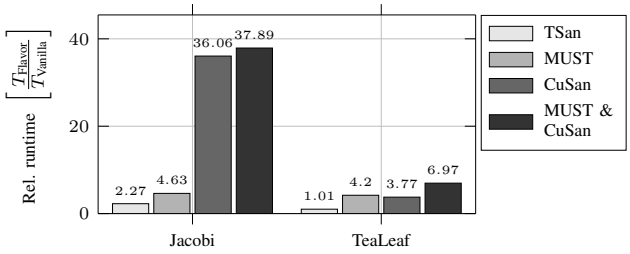

Fig. 10. The relative runtime overhead w.r.t. vanilla. Vanilla runtime: 1.35 s and 0.75 s for Jacobi and TeaLeaf, respectively.

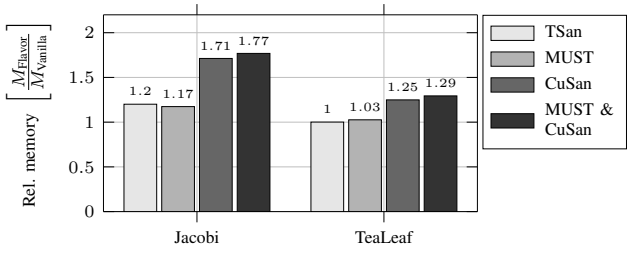

Fig. 11. The relative memory overhead of a single MPI process w.r.t. vanilla. Vanilla RSS: 311 MB and 283 MB for Jacobi and TeaLeaf, respectively.

is almost at the level of vanilla. MUST's startup cost adds some overhead due to the low runtime of the problem model. MUST & CuSan slows down the execution about 7× as fibers for both non-blocking MPI and CUDA are required, adding more overhead to the execution.

TABLE I shows event counters reported by CuSan w.r.t. intercepted CUDA calls and their resulting TSan-related API calls for a process. A fiber is created per CUDA stream. Per kernel call, multiple pointers may be annotated as read or write using TSan. *AnnotateHappensBefore* events occur more often than *AnnotateHappensAfter* as we handle default stream semantics of kernel calls and implicit synchronization points like `cudaMemcpy`. Starting a happens-before arc on a kernel with default stream also starts one for each other stream (as default operations block all succeeding stream operations until the kernel is completed). Likewise, this applies to, e.g., `cudaMemcpy` as it is executed on the default stream and, hence, has the same default stream synchronization semantics. The terminating happens-after events occur for (i) operations on the default stream that have an implicit barrier for all other streams (see Fig. 3), (ii) Synchronization calls (iii) and Memcpy operations. This is obvious for TeaLeaf: as it only uses the default stream the first does not apply, and hence it has 632 happens-after events which is the number of Memcpy and Synchronization calls. The `tsan_read/write_range` sizes are significantly larger with Jacobi compared to TeaLeaf. This is the leading factor for performance overhead, and is discussed in more detail in Section V-B.

TABLE I
JACOBI AND TEALEAF CUDA AND TSAN RUNTIME EVENT COUNTERS FOR ONE MPI PROCESS AS REPORTED BY CUSAN.

	Metric	Jacobi	TeaLeaf
CUDA	Stream	2	1
	Memset	2	36
	Memcpy	602	102
	Synchronization calls	900	530
	Kernel calls	1,200	767
TSan	Switch To Fiber	3,622	1,882
	AnnotateHappensBefore	1,804	905
	AnnotateHappensAfter	1,515	632
	Memory Read Range	2,102	623
	Memory Write Range	2,403	1,074
	Memory Read Size [avg KB]	19,705.62	15.98
	Memory Write Size [avg KB]	16,421.35	17.58

2) Memory: Fig. 11 shows the induced memory overhead of the correctness tools. To that end, we query the resident set size (RSS) at the invocation time of `MPI_Finalize`.

Here, CuSan adds most memory overhead as the tracking of CUDA device pointers within TSan represents the majority of memory usage for these CUDA-aware MPI applications.

B. Jacobi Solver Scaling Test

CuSan has an overhead factor of about 36× for the Jacobi solver for our tested model size. This is due to the high amount of memory that is tracked for each kernel invocation. In our testing, completely removing memory annotations but keeping the rest of our instrumentation brings the overhead down to almost vanilla. Currently, for each kernel call, we must annotate the access semantics of the whole device pointers memory range with TSan for our data race analysis. The more memory tracked with TSan, the more runtime overhead is induced, see Fig. 12. As illustrated, runtime overhead of CuSan scales approximately with the amount of memory that is tracked by TSan (due to our CUDA annotations). Reducing the tracked memory by, e.g., reducing the domain size, hence reduces runtime overhead.

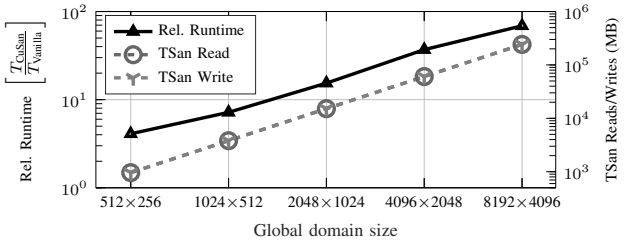

Fig. 12. Jacobi relative runtime overhead w.r.t. vanilla based on the global domain size. The right y-axis shows the total sum of tracked memory access operations for two MPI processes, i.e., total bytes tracked with `tsan_write_range` and `tsan_read_range` calls, respectively.

VI. DISCUSSION

We discuss primary limitations of CuSan. A particular challenge is to verify behavior of the CUDA API relevant to data race detection. To that end, our current scale of CUDA API support is limited to a manually verified set w.r.t. synchronization behavior, see Section III-B.

A. CUDA API Coverage

The CuSan implementation focuses on explicit synchronization and key implicit synchronization points (like memory copies) and the default stream semantics for CUDA-aware MPI, see Section III. Our implementation is currently based on documentation of CUDA 11.5, with implicit synchronization behavior verified by our test suite, see Section VI-C. Others have noted unclear documentation or undocumented behaviors with CUDA synchronization [40]. Therefore, it is crucial to verify particular concurrency behavior for each supported CUDA feature within CuSan. Future work aims to extend the synchronization model to cover a broader range of CUDA API calls based on practical usage patterns of other CUDA-aware MPI applications.

B. Per-thread Default Stream Support

CuSan currently assumes legacy default stream semantics and single-threaded host code execution. Future work will explicitly support both legacy and per-thread default stream modes. The latter provides each host thread with a default stream that does not have the same blocking characteristics as with legacy. However, using a mix of per-thread default stream and legacy default stream would still have the same blocking characteristics [28, Section 3]: "The per-thread default stream is not a non-blocking stream and will synchronize with the legacy default stream if both are used in a program.". In addition, explicit synchronization semantics, such as `cudaDeviceSynchronize` may only block on already submitted kernels on streams. While one thread calls the synchronization, other threads could still submit work on streams that start before the synchronization finishes [40].

C. Correctness Test Suite

While our focus was primarily on the technical aspects of CUDA-related race detection, we developed a test suite containing small-scale codes used to evaluate CuSan. This test suite includes both manually verified correct and incorrect (i.e., containing data races) examples of CUDA-aware MPI usage. The purpose is to create a (i) test harness to verify CuSan's race detection capabilities, and (ii) feature documentation, to demonstrate supported CUDA features and their particular behavior as described in the previous sections. Hence, for now, all tests are correctly classified by CuSan. The code is available at https://github.com/tudasc/cusan-tests/tree/main/testsuite.

D. CuSan Overhead

Tracking CUDA device pointer semantics with TSan introduces significant overhead (around 36× for the Jacobi solver evaluation). While TSan overhead is stated to be typically around 5×–15× [21], higher overheads have been reported for other scenarios [15], [22], [41], [42]. In Section V-B, we have shown that overhead in CuSan directly correlates with the amount of memory access tracking. In the future, CuSan could implement analyses to limit memory access tracking by identifying and focusing on the boundary regions of data exchanged via MPI, rather than tracking entire device pointer allocations.

E. CUDA Correctness Tools

A comparable approach to CuSan exists within PyTorch [43], focused on detecting unsynchronized tensor accesses. It tracks stream synchronizations and tensor memory accesses, analyzing the synchronization state for potential data races at each kernel launch. However, this effort is limited to the PyTorch Python API and CUDA-only races. In contrast, CuSan can find races in C and C++ codes, including unsynchronized CUDA managed memory access, or CUDA-aware MPI applications (in combination with MUST). Other correctness tools focus on identifying errors like kernel thread-level synchronization issues or memory leaks [44], [45].

VII. Conclusion

For efficiency, MPI is combined with node-level parallel programming models like CUDA. CUDA-aware MPI libraries facilitate direct communication of device memory, eliminating the need for manual copying. However, this hybrid model adds complexity and presents significant correctness challenges, as both CUDA and MPI operate asynchronously w.r.t. the host. In such models, data-dependent operations between asynchronous MPI and CUDA calls require careful synchronization by the user. Users must synchronize CUDA operations before initiating dependent MPI calls and also ensure that non-blocking MPI calls are completed before starting related CUDA operations. This is error-prone.

To address these challenges, this paper introduced CuSan, a tool that enables synchronization detection in CUDA-aware MPI applications. CuSan uses ThreadSanitizer (TSan) as the underlying data race detector, exposing CUDAs particular memory access and synchronization semantics to its annotation API. The inherent concurrency of CUDA-stream based executions are modelled using TSan fibers. To find concurrency issues with non-blocking MPI communication, the dynamic MPI correctness checker MUST already integrates with TSan fibers. With CuSan and MUST, the concurrency semantics of CUDA streams and non-blocking MPI communications is properly exposed to TSan for its data race analysis.

Our evaluation on two mini-apps shows runtime overhead factors of CuSan ranging from 6× to 36×, depending on the model size. The overhead is directly related to the amount of memory marked for access semantics tracking in TSan. Hence, smaller model runs should be preferred for data race analysis. Memory overhead is consistently below 1.8×.

In summary, we believe CuSan aids users develop data-race-free CUDA-aware MPI applications. In the future, we will extend CUDA API support and also look into other accelerator models. CuSan is available at https://github.com/tudasc/cusan.

Acknowledgment

This work has received funding from the Federal Ministry of Education and Research (BMBF) and the state of North Rhine-Westphalia as part of the NHR program.

References

[1] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 4.1*, Nov. 2023, accessed: 2024-08-01. [Online]. Available: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf

[2] I. Laguna, R. Marshall, K. Mohror, M. Ruefenacht, A. Skjellum, and N. Sultana, "A Large-Scale Study of MPI Usage in Open-Source HPC Applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. ACM, 2019.

[3] A. Hück, T. Jammer, J. Protze, and C. Bischof, "Investigating the Usage of MPI at Argument-Granularity in HPC Codes," in *Proceedings of EuroMPI2023: the 30th European MPI Users' Group Meeting*, ser. EuroMPI2023. ACM, 2023, pp. 1–10.

[4] NVIDIA, "NVIDIA CUDA Toolkit," https://developer.nvidia.com/cuda-toolkit, accessed: 2024-08-01.

[5] A. Droste, M. Kuhn, and T. Ludwig, "MPI-checker: static analysis for MPI," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM '15. ACM, 2015.

[6] J. Vetter and B. de Supinski, "Dynamic Software Testing of MPI Applications with Umpire," in *SC '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, 2000, pp. 51–51.

[7] M. Laurent, E. Saillard, and M. Quinson, "The MPI Bugs Initiative: a Framework for MPI Verification Tools Evaluation," in *IEEE/ACM 5th International Workshop on Software Correctness for HPC Applications (Correctness)*, 2021, pp. 1–9.

[8] J.-P. Lehr, T. Jammer, and C. Bischof, "MPI-CorrBench: Towards an MPI Correctness Benchmark Suite," in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC'21. ACM, 2021, pp. 69–80.

[9] M. Süß and C. Leopold, "Common mistakes in OpenMP and how to avoid them," in *International Workshop on OpenMP*. Springer, 2005, pp. 312–323.

[10] J. F. Münchhalfen, T. Hilbrich, J. Protze, C. Terboven, and M. S. Müller, "Classification of Common Errors in OpenMP Applications," in *Using and Improving OpenMP for Devices, Tasks, and More*. Springer, 2014, pp. 58–72.

[11] T. Jammer, A. Hück, J.-P. Lehr, J. Protze, S. Schwitanski, and C. Bischof, "Towards a Hybrid MPI Correctness Benchmark Suite," in *Proceedings of the 29th European MPI Users' Group Meeting*, ser. EuroMPI/USA '22. ACM, 2022, pp. 46–56.

[12] E. Saillard, P. Carribault, and D. Barthou, "PARCOACH: Combining static and dynamic validation of MPI collective communications," *The International Journal of High Performance Computing Applications*, vol. 28, no. 4, pp. 425–434, 2014.

[13] T. Hilbrich, M. Schulz, B. R. de Supinski, and M. S. Müller, "MUST: A scalable approach to runtime error detection in MPI programs," in *Tools for high performance computing 2009*. Springer, 2010, pp. 53–66.

[14] Intel, "Intel Trace Analyzer and Collector," https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/trace-analyzer.html, 2023, accessed: 2024-08-01.

[15] J. Protze, M. Schulz, D. H. Ahn, and M. S. Müller, "Thread-local concurrency: a technique to handle data race detection at programming model abstraction," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '18. ACM, 2018, pp. 144–155.

[16] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D. K. Panda, "GPU-Aware MPI on RDMA-Enabled Clusters: Design, Implementation and Evaluation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 10, pp. 2595–2605, 2014.

[17] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, 2004, pp. 97–104.

[18] D. K. Panda, H. Subramoni, C.-H. Chu, and M. Bayatpour, "The MVAPICH project: Transforming research into high-performance MPI library for HPC community," *Journal of Computational Science*, vol. 52, 2021, case Studies in Translational Computer Science.

[19] A. M. Aji, P. Balaji, J. Dinan, W.-c. Feng, and R. Thakur, "Synchronization and Ordering Semantics in Hybrid MPI+GPU Programming," in *2013 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum*, 2013, pp. 1020–1029.

[20] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: data race detection in practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, ser. WBIA '09. ACM, 2009, pp. 62–71.

[21] The LLVM Project, "ThreadSanitizer," https://clang.llvm.org/docs/ThreadSanitizer.html, accessed: 2024-08-01.

[22] J. Jenke, S. Schwitanski, I. Thärigen, and M. S. Müller, "Mapping High-Level Concurrency from OpenMP and MPI to ThreadSanitizer Fibers," in *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W '23. ACM, 2023, pp. 187–195.

[23] A. Hück, J.-P. Lehr, S. Kreutzer, J. Protze, C. Terboven, C. Bischof, and M. S. Müller, "Compiler-aided type tracking for correctness checking of MPI applications," in *IEEE/ACM 2nd Intl. Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE, 2018, pp. 51–58.

[24] A. Hück, J. Protze, J. P. Lehr, C. Terboven, C. Bischof, and M. S. Müller, "Towards compiler-aided correctness checking of adjoint MPI applications," in *IEEE/ACM 4th Intl. Workshop on Software Correctness for HPC Applications (Correctness)*, 2020, pp. 40–48.

[25] A. Hück, S. Kreutzer, J. Protze, J.-P. Lehr, C. Bischof, C. Terboven, and M. S. Müller, "Compiler-Aided Type Correctness of Hybrid MPI-OpenMP Applications," *IT Professional*, vol. 24, no. 2, pp. 45–51, 2022.

[26] The LLVM Project, "Fiber support for thread sanitizer," https://reviews.llvm.org/D54889, accessed: 2024-08-01.

[27] S. Atzeni, G. Gopalakrishnan, Z. Rakamaric, D. H. Ahn, I. Laguna, M. Schulz, G. L. Lee, J. Protze, and M. S. Müller, "ARCHER: Effectively Spotting Data Races in Large OpenMP Applications," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 53–62.

[28] NVIDIA, "NVIDIA CUDA Runtime API 11.5," https://docs.nvidia.com/cuda/archive/11.5.0/cuda-runtime-api/, accessed: 2024-08-01.

[29] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, "GPU scheduling on the NVIDIA TX2: Hidden details revealed," in *2017 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2017, pp. 104–115.

[30] NVIDIA, "NVIDIA CUDA C Programming Guide 11.5," https://docs.nvidia.com/cuda/archive/11.5.0/cuda-c-programming-guide/index.html, accessed: 2024-08-01.

[31] ——, "CUDA Driver API Documentation 11.5," https://docs.nvidia.com/cuda/archive/11.5.0/cuda-driver-api/index.html, accessed: 2024-08-01.

[32] A. M. Aji, L. S. Panwar, F. Ji, K. Murthy, M. Chabbi, P. Balaji, K. R. Bisset, J. Dinan, W.-c. Feng, J. Mellor-Crummey, X. Ma, and R. Thakur, "MPI-ACC: Accelerator-Aware MPI for Scientific Applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1401–1414, 2016.

[33] H. Zhou, K. Raffenetti, Y. Guo, and R. Thakur, "MPIX Stream: An Explicit Solution to Hybrid MPI+X Programming," in *Proceedings of the 29th European MPI Users' Group Meeting*, ser. EuroMPI/USA '22. ACM, 2022, pp. 1–10.

[34] N. Dryden, N. Maruyama, T. Moon, T. Benson, A. Yoo, M. Snir, and B. Van Essen, "Aluminum: An Asynchronous, GPU-Aware Communication Library Optimized for Large-Scale Training of Deep Neural Networks on HPC Systems," in *2018 IEEE/ACM Machine Learning in HPC Environments (MLHPC)*, 2018, pp. 1–13.

[35] J. Choi, Z. Fink, S. White, N. Bhat, D. F. Richards, and L. V. Kale, "GPU-aware Communication with UCX in Parallel Programming Models: Charm++, MPI, and Python," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2021, pp. 479–488.

[36] M. Moraru, A. Roussel, M. Pérache, H. Taboada, C. Jaillet, and M. Krajecki, "Benefits of MPI Sessions for GPU MPI applications," in *EuroMPI'21-28th European MPI Users' Group Meeting*, 2021.

[37] The LLVM Project, "Compiling CUDA with clang," https://llvm.org/docs/CompileCudaWithLLVM.html, accessed: 2024-08-01.

[38] NVIDIA, "Jacobi Solver," https://github.com/NVIDIA-developer-blog/code-samples/tree/master/posts/cuda-aware-mpi-example/src, accessed: 2024-08-01.

[39] UoB-HPC, "TeaLeaf," https://github.com/UoB-HPC/TeaLeaf, accessed: 2024-08-01.

[40] M. Yang, N. Otterness, T. Amert, J. Bakita, J. H. Anderson, and F. D. Smith, "Avoiding Pitfalls when Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems," in *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, vol. 106, 2018, pp. 20:1–20:21.

[41] J. Protze, I. Thärigen, and J. Wahle, "Understanding the Performance of Dynamic Data Race Detection," in *2021 IEEE/ACM 5th International Workshop on Software Correctness for HPC Applications (Correctness)*, 2021, pp. 33–40.
[42] S. Schwitanski, J. Jenke, F. Tomski, C. Terboven, and M. S. Müller, "On-the-Fly Data Race Detection for MPI RMA Programs with MUST," in *2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness)*, 2022, pp. 27–36.
[43] PyTorch, "CUDA Stream Sanitizer," https://pytorch.org/docs/stable/cuda._sanitizer.html, accessed: 2024-08-01.
[44] NVIDIA, "Compute Sanitizer," https://docs.nvidia.com/compute-sanitizer/ComputeSanitizer/index.html, accessed: 2024-08-01.
[45] M. Tarek Ibn Ziad, S. Damani, A. Jaleel, S. W. Keckler, and M. Stephenson, "CuCatch: A Debugging Tool for Efficiently Catching Memory Safety Violations in CUDA Applications," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 124–147, 2023.

ARTIFACT DESCRIPTION

The following descriptions explain how to set up our CuSan and MUST toolchain for evaluation. Our test environment is available at: https://github.com/tudasc/cusan-tests

Software and Hardware Requirements

To evaluate CuSan, we tested these software packages:

- CUDA toolkit 11.5 and 11.8
- CUDA-aware OpenMPI 4.1.4 and 4.1.6
- Clang/LLVM 14.0.6, Python 3.10, Git 2.40, CMake 3.20

Our test suite, see Section VI-C, additionally needs:

- llvm-lit (available with `pipx`)
- FileCheck binary (comes with LLVM)

The following GPUs were evaluated for compatibility NVIDIA Tesla T4, NVIDIA Tesla V100 and NVIDIA Tesla H100. Each code was compiled with `arch=sm_70`.

ThreadSanitizer Suppressions: When using TSan with these libraries, false positive may occur. To that end, we use suppression lists for TSan that avoid these. Our custom suppression lists are tailored to Lichtenberg (TU Darmstadt) and CLAIX-2023 (RWTH Aachen) HPC clusters. Unfortunately, they may require changes for any particular HPC cluster.

Lichtenberg and CLAIX modules: We provide a list of manually loaded modules for each cluster, other dependencies are available in `PATH`. Lichtenberg, followed by CLAIX:

```
1) gcc/11.2.0   3) openmpi/4.1.6   5) python/3.10.10
2) cuda/11.8    4) git/2.40.0      6) clang/14.0.6

1) GCCcore/.11.3.0 8) libpciaccess/0.16 15) OpenMPI/4.1.4
2) zlib/1.2.12     9) hwloc/2.7.1       16) gompi/2022a
3) binutils/2.38   10) OpenSSL/1.1      17) CUDA/11.6.0
4) GCC/11.3.0      11) libevent/2.1.12  18) GDRCopy/2.3
5) numactl/2.0.14  12) UCX/1.12.1       19) UCX-CUDA/1.12.1-
6) XZ/5.2.5        13) PMIx/4.1.2           CUDA-11.6.0
7) libxml2/2.9.13  14) UCC/1.0.0        20) clang/14-release
```

Prerequisites for the Test Environment

After the test environment repository was cloned, we need to set up CuSan and MUST as prerequisites. To that end, we provide scripts that download and compile these codes automatically. The scripts expect the appropriate software modules to be available/loaded. At the end, environment variables, as per the scripts direction, have to be set for MUST and CuSan. These will be used by the test environment initialization.

```
$ git clone --branch v1.0 https://github.com/tudasc/cusan-tests
$ cd cusan-tests
# Setup CuSan, the script will instruct how to set CUSAN_PATH:
$ ./support/cusan-bootstrap.sh
$ export CUSAN_PATH=...
# Setup MUST, the script will instruct how to set MUST_PATH:
$ ./support/must-bootstrap.sh
$ export MUST_PATH=...
```

Initializing the Test Environment

With the prerequisites installed, the actual test environment can be initialized. We use CMake for this.

```
# Assume we are in root folder of cusan-tests.
$ mkdir build && cd build
# Setup scripts for evaluation, should detect CuSan and MUST:
$ cmake ..
# Build the mini-apps. Vanilla, TSan, CuSan versions:
$ make jacobi-all-build
$ make tealeaf-all-build
```

Executing the Test Suite

We have a test suite consisting of unit tests with correct and incorrect usage of CUDA. These tests are executed with llvm-lit and FileCheck.

```
# Assume we are in the build folder
$ make check-cutests
```

This should print (unordered) output like:

```
PASS: CuSanTest :: cuda-to-mpi/send_mca_user_malloc_w_r.c (1 of 49)
PASS: CuSanTest :: cuda-to-mpi/send_ds_def_managed_r_r.c (2 of 49)
PASS: CuSanTest :: cuda-to-mpi/send_mca_user_malloc_w_r_nok.c (3 of 49)
...
```

Executing the Mini-Apps

The mini-apps Jacobi and TeaLeaf can be executed directly on a node or submitted to the Slurm workload manager. For the latter, our batch scripts are only implemented for the particularities of the Lichtenberg cluster at TU Darmstadt and, thus, need adaption for other systems.

Executing directly on the node: We assume that the test environment was initialized. The following illustrates execution with Jacobi (analogous for TeaLeaf).

```
# Assume we are in the build folder, vanilla:
$ make jacobi-vanilla-run
# Run MUST & TSan instrumented version
$ make jacobi-must-run
# Run CuSan instrumented version
$ make jacobi-run
```

Executing with Slurm: The initialization step generated *make* targets to submit to Slurm. However, the scripts need to be adapted for a particular HPC cluster. To that end, go to each mini-app folder and adapt `sbatch.sh` and `sbatch-scale.sh` for your cluster environment.

```
# Assume we are in the build folder
# Benchmark runtime and memory
$ make jacobi-sbatch
$ make tealeaf-sbatch
# Benchmark scaling for different domain sizes
$ make jacobi-scale
```