NVIDIA

SANITIZER API

v2022.4.1 | January 2023

Reference Manual

TABLE OF CONTENTS

Chapter 1. INTRODUCTION

1.1. Overview

The Compute Sanitizer API enables the creation of sanitizing and tracing tools that target CUDA applications. Examples of such tools are memory and race condition checkers. The Compute Sanitizer API is composed of three APIs: the callback API, the patching API and the memory API. It is delivered as a dynamic library on supported platforms.

Chapter 2. USAGE

2.1. Compatibility and Requirements

The Compute Sanitizer tools require CUDA 11.0 or newer.

The Compute Sanitizer API requires CUDA 10.1 or newer. Compute Sanitizer API calls will fail with **`SANITIZER_ERROR_NOT_INITIALIZED`** if the CUDA driver version is not compatible with the Compute Sanitizer version.

2.2. Callback API

The Compute Sanitizer Callback API allows you to register a callback into user code. The callback is invoked when the application calls a CUDA runtime or driver function, or when certain events occur in the CUDA driver. The following terminology is used by the Callback API.

- **Callback domain:** Callbacks are grouped into domains to make it easier to associate callback functions with groups of related CUDA functions or events. The following callback domains are defined by **`Sanitizer_CallbackDomain`**.
    1. CUDA driver functions
    2. CUDA runtime functions
    3. CUDA resource tracking
    4. CUDA synchronization notification
    5. CUDA grid launches
    6. CUDA memcpy operations
    7. CUDA memset operations
    8. CUDA batch memory operations
- **Callback ID:** Each callback is given a unique ID within the corresponding callback domain in order to identify it within the callback function. The CUDA driver API IDs are defined in **`sanitizer_driver_cbid.h`** and the CUDA runtime API IDs are defined in **`sanitizer_runtime_cbid.h`**. Other callback IDs are

defined in **sanitizer_callbacks.h**. All of these headers are included as part of **sanitizer.h**.

- **Callback Function:** The callback function must be of the type **Sanitizer_CallbackFunc**. This function type has two arguments that specify the callback: the domain and the ID that identifies why the callback is occurring. The type also has a **cbdata** argument that is used to pass data specific to the callback.
- **Subscriber:** A subscriber is used to associate each of the callback functions with one or more CUDA API functions. There can be at most one subscriber initialized with **sanitizerSubscribe** at any time. Before initializing a new subscriber, the existing one must be finalized with **sanitizerUnsubscribe** .

The subscriber should be initialized prior to making any CUDA API call to ensure correctness of the reported data.

Each callback domain is described in detail below. Unless explicitly stated, it is not supported to call any CUDA runtime or driver API from within a callback function. Doing so may cause the application to hang. However, it is supported to call Compute Sanitizer Memory APIs from within callback functions.

2.2.1. Driver and Runtime API Callbacks

Using the Callback API with the **SANITIZER_CB_DOMAIN_DRIVER_API** or **SANITIZER_CB_DOMAIN_RUNTIME_API** domains, a callback function can be associated with one or more CUDA API functions. When those CUDA functions are called in the application, the callback function is invoked as well. For these domains, the **cbdata** argument to the callback function will be of the type **Sanitizer_CallbackData**.

You can call **cudaDeviceSynchronize**, **cudaStreamSynchronize**, **cuCtxSynchronize** and **cuStreamSynchronize** from within a driver or runtime API callback function.

The following code shows a typical sequence used to associate a callback function with one or more CUDA API functions. For simplicity, error checking code was removed.

```
Sanitizer_SubscriberHandle handle;
MyDataStruct *my_data = ...;
...
sanitizerSubscribe(&handle, my_callback, my_data);
sanitizerEnableDomain(1, handle, SANITIZER_CB_DOMAIN_RUNTIME_API);
```

First, **sanitizerSubscribe** is used to initialize a subscriber with the **my_callback** callback function. Next, **sanitizerEnableDomain** is used to associate that callback with all the CUDA runtime functions. Using this code sequence will cause **my_callback** to be called twice each time any of the CUDA runtime API functions are invoked, once on entry to the CUDA function and once just before the CUDA function exits. Compute Sanitizer callback API functions **sanitizerEnableCallback** and **sanitizerEnableAllDomains** can also be used to associate CUDA API functions with a callback.

The following code shows a typical callback function.

```
void SANITIZERAPI
my_callback(void *userdata,
            Sanitizer_CallbackDomain domain,
            Sanitizer_CallbackId cbid,
            const void *cbdata)
{
    const Sanitizer_CallbackData *cbInfo = (Sanitizer_CallbackData *)cbdata;
    MyDataStruct *my_data = (MyDataStruct *)userdata;

    if ((domain == SANITIZER_CB_DOMAIN_RUNTIME_API) &&
        (cbid == SANITIZER_RUNTIME_TRACE_CBID_cudaMemcpy_v3020) &&
        (cbInfo->callbackSite == SANITIZER_API_ENTER))
    {
        cudaMemcpy_v3020_params *funcParams = (cudaMemcpy_v3020_params *)
(cbInfo->functionParams);
        size_t count = funcParams->count;
        enum cudaMemcpyKind kind = funcParams->kind
        ...
    }
    ...
}
```

In the callback function, **Sanitizer_CallbackDomain** and **Sanitizer_CallbackId** parameters can be used to determine which CUDA API function invocation is triggering this callback. In the example above, we are checking for the CUDA runtime **cudaMemcpy** function. The **cbdata** parameter holds a structure of useful information that can be used within the callback. In this case, we use the **callbackSite** member of the structure to detect that the callback is occurring on entry to **cudaMemcpy**, and we use the **functionParams** member to access the parameters to **cudaMemcpy**. To access the parameters, we first cast **functionParams** to a structure type corresponding to the **cudaMemcpy** function. These parameter structures are contained in **generated_cuda_runtime_api_meta.h**, **generated_cuda_meta.h** and a number of other files.

2.2.2. Resource Callbacks

Using the Callback API with the **SANITIZER_CB_DOMAIN_RESOURCE** domain, a callback function can be associated with some CUDA resource creation and destruction events. For example, when a CUDA context is created, the callback function is invoked with a callback ID equal to **SANITIZER_CBID_RESOURCE_CONTEXT_CREATED**. For this domain, the **cbdata** argument is one of the following types:

- **Sanitizer_ResourceContextData** for CUDA context creation and destruction
- **Sanitizer_ResourceStreamData** for CUDA stream creation and destruction
- **Sanitizer_ResourceModuleData** for CUDA module load and unload
- **Sanitizer_ResourceMemoryData** for CUDA memory allocation and de-allocation

2.2.3. Synchronization Callbacks

Using the Callback API with the **SANITIZER_CB_DOMAIN_SYNCHRONIZE** domain, a callback function can be associated with CUDA context and stream synchronizations. For example, when a CUDA context is synchronized, the callback function is invoked

with a callback ID equal to **SANITIZER_CBID_SYNCHRONIZE_CONTEXT_SYNCHRONIZED**. For this domain, the **cbdata** argument is of the type **Sanitizer_SynchronizeData**.

2.2.4. Launch Callbacks

Using the Callback API with the **SANITIZER_CB_DOMAIN_LAUNCH** domain, a callback function can be associated with CUDA kernel launches. For example, when a CUDA kernel launch has started, the callback function is invoked with a callback ID equal to **SANITIZER_CBID_LAUNCH_BEGIN**. For this domain, the **cbdata** argument is of the type **Sanitizer_LaunchData**.

2.2.5. Memcpy Callbacks

Using the Callback API with the **SANITIZER_CB_DOMAIN_MEMCPY** domain, a callback function can be associated with CUDA memcpy operations. For example, when a **cudaMemcpy** API function is called, the callback function is invoked with a callback ID equal to **SANITIZER_CBID_MEMCPY_STARTING**. For this domain, the **cbdata** argument is of the type **Sanitizer_MemcpyData**.

2.2.6. Memset Callbacks

Using the Callback API with the **SANITIZER_CB_DOMAIN_MEMSET** domain, a callback function can be associated with CUDA memset operations. For example, when a **cudaMemset** API function is called, the callback function is invoked with a callback ID equal to **SANITIZER_CBID_MEMSET_STARTING**. For this domain, the **cbdata** argument is of the type **Sanitizer_MemsetData**.

2.2.7. Batch Memory Operations Callbacks

Using the Callback API with the **SANITIZER_CB_DOMAIN_BATCH_MEMOP** domain, a callback function can be associated with CUDA batch memory operations. For example, when a **cuStreamWriteValue** API function is called, the callback function is invoked with a callback ID equal to **SANITIZER_CBID_BATCH_MEMOP_WRITE**. For this domain, the **cbdata** argument is of the type **Sanitizer_BatchMemopData**.

2.3. Patching API

The Compute Sanitizer Patching API allows you to load patch functions and insert them into user code. Patch functions will be invoked when the application's CUDA code executes certain instructions or calls certain CUDA device functions. The following terminology is used by the Patching API:

- **Instruction ID**: Each patchable event is given a unique ID than can be passed to patching API functions to specify that these events should be patched. Instruction IDs are defined by **Sanitizer_InstructionId**.
- **Instrumentation point**: A location in the original CUDA code that is being instrumented by the Compute Sanitizer API. Upon execution, the user code path is

modified so that a patch gets executed either before or after the patched event. All patches are executed prior to the event, with the exception of device-side malloc.

- **Patch**: A CUDA **__device__** function that the Compute Sanitizer will insert into another existing CUDA code. Patch function signatures must match the one expected by the API (see below for the expected signature types).

2.3.1. Writing a Patch

The patch must follow the function signature required by the Compute Sanitizer API for a given instruction ID. The mapping of instruction ID to function signature is documented in the comments of **Sanitizer_InstructionId** in **sanitizer_patching.h**. For instance, if we wish to patch a memory access using the instruction ID **SANITIZER_INSTRUCTION_MEMORY_ACCESS**, we need to use the **SanitizerCallbackMemoryAccess** type.

```
extern "C" __device__
SanitizerPatchResult SANITIZERAPI my_memory_access_callback(
    void* userdata,
    uint64_t pc,
    void* ptr,
    uint32_t accessSize,
    uint32_t flags)
{
    MyDeviceDataStruct *my_data = (MyDeviceDataStruct *)userdata

    if ((flags & SANITIZER_MEMORY_DEVICE_FLAG_WRITE) != 0)
        // log write
    else
        // log read

    return SANITIZER_PATCH_SUCCESS;
}
```

In this patch, we log write and read accesses to a structure we allocated previously. **extern "C"** ensures that the patch name will not be mangled, allowing us to use its name as a string directly in calls to **sanitizerPatchInstructions** (see below).

There can be multiple patches defined in a single CUDA file. This file must then be compiled using the following nvcc options:

```
$ nvcc --cubin --compile-as-tools-patch MySanitizerPatches.cu -o
 MySanitizerPatches.cubin
```

The **--cubin** option can be replaced by **--fatbin** if a fatbin is preferred over a cubin as the output file.

2.3.2. Insert a Patch

Once the patch has been generated, it can be inserted into user code by using the following procedure:

1. **Load the patch**. There are two APIs used to load the patch: **sanitizerAddPatchesFromFile** and **sanitizerAddPatches**. They use the same input format as **cuModuleLoad** and **cuModuleLoadData**, respectively.

2. **Select which instructions to patch** by using the **sanitizerPatchInstructions** API.
3. **Patch user code** by using the **sanitizerPatchModule** API.
4. Optionally, **set the callback data for patches** by using the **sanitizerSetCallbackData** API.

The following code shows a typical sequence using these APIs. For simplicity, error checking was removed.

```
CUcontext ctx = ... // current CUDA context
sanitizerAddPatchesFromFile("MySanitizerPatches.cubin", ctx);

CUmodule module = ... // module containing the user code
sanitizerPatchInstructions(SANITIZER_INSTRUCTION_MEMORY_ACCESS, module,
 "my_memory_access_callback");

sanitizerPatchModule(module);

MyDeviceDataTracker *deviceDataTracker;
cudaMalloc(&deviceDataTracker, sizeof(*deviceDataTracker));

CUfunction function = ... // kernel to be launched for which we want to set the
 callbackdata for the patches
sanitizerSetCallbackData(function, deviceDataTracker);
```

All subsequent launches using code from this CUDA module will be instrumented and **my_memory_access_callback** will be invoked before every memory access. However, the callback data is only set for all subsequent launches of the given kernel. An easy way to have a kernel **CUfunction**, is through the Sanitizer launch callbacks. Instrumentation can be removed by using the **sanitizerUnpatchModule** API.

2.4. Memory API

The Compute Sanitizer Memory API provides replacement functions for the CUDA Memory API that can be safely called from within Compute Sanitizer callbacks.

- **sanitizerAlloc** is a replacement for **cudaMalloc**.
- **sanitizerFree** is a replacement for **cudaFree**.
- **sanitizerMemcpyHostToDeviceAsync** is a replacement for **cudaMemcpyAsync** for host-to-device copies.
- **sanitizerMemcpyDeviceToHost** is a replacement for **cudaMemcpy** for device-to-host copies.
- **sanitizerMemset** is a replacement for **cudaMemset**.

These functions can also be called in normal user code, where they can be mixed with the CUDA API. For instance, memory allocated with **sanitizerAlloc** can be freed with **cudaFree**. However, since only CUDA API calls will cause callbacks to be invoked, this can lead to an incoherent tracking state and should be avoided.

When called from a launch domain callback, **sanitizerMemcpyDeviceToHost** may only be used with host pinned memory as destination.

Chapter 3. LIMITATIONS

No known issues at this time.

Notice

Trademarks

Copyright