NVTX API FOR COMPUTE SANITIZER

v2023.1.1 | April 2023

Reference Manual

TABLE OF CONTENTS

Chapter 1. INTRODUCTION

1.1. Overview

The NVTX Memory API for Compute Sanitizer allows CUDA programs to notify Compute Sanitizer about memory restrictions: memory pools management or permissions restrictions, in addition to memory labeling. The tools are notified through NVTX (NVIDIA Tools Extension), a header-only C library used by various NVIDIA tools. Latest NVTX headers can be downloaded on our GitHub repository (experimental branch).

This API has the following main goals:

- Programs can mark allocations as memory pools, allowing Compute Sanitizer to be aware of which parts of this specific allocation are actually used. When using the Memcheck tool, you are notified if unregistered parts of the pool are accessed by the program, errors that could have been missed otherwise. When using the Initcheck tool, in combination with option **`--track-unused-memory yes`**, you are not notified for unused memory in non-registered regions, therefore avoiding false positives.
- Programs can label allocations with meaningful names, allowing you to identify an allocation associated to a specific error by its name (e.g., allocation that is leaking, or unused).
- Programs can restrict some allocations to a specific set of permissions (e.g., read-only or write-only) applicable for a specific scope (e.g., CUDA stream, device or whole program). When using the Memcheck tool, violation of these restrictions will result in an error.

Chapter 2. USAGE

2.1. Compatibility and Requirements

The Compute Sanitizer tools require CUDA 11.0 or newer.

The NVTX Memory API is supported by Compute Sanitizer starting CUDA 11.3, using the **`--nvtx yes`** option. Starting CUDA 12.0, this option is enabled by default.

Compute Sanitizer requires the CUDA runtime to be initialized before calling NVTX.

```
// NVTX calls are not allowed before CUDA runtime initialization.

// Forces CUDA runtime initialization.
cudaFree(0);

// NVTX calls are now allowed.
```

NVTX structures must be zero-initialized. Examples on this page use C++ empty initializer (**`{}`**). If you are using C, you can use **`memset`** or use the intializer syntax with at least one field (C does not support empty initalizers).

2.2. NVTX Domain

All NVTX calls requires you to create a NVTX domain. This can be achieved using **`nvtxDomainCreateA`**.

```
// Requires <nvtx3/nvToolsExt.h>

auto nvtxDomain = nvtxDomainCreateA("my-domain");
```

For now, NVTX domains have no specific usage, but will have one in a future Compute Sanitizer version.

2.3. Suballocation API

2.3.1. Pools Management

Any allocation created with **cudaMalloc** can be registered as a memory pool using **nvtxMemHeapRegister**. The following code example allocates 64 bytes and registers the allocation as a memory pool.

```
// Requires <nvtx3/nvToolsExtMem.h>
// (see https://github.com/NVIDIA/NVTX/tree/dev-mem-api/c/include)

void *ptr;
cudaMalloc(&ptr, 64);

nvtxMemVirtualRangeDesc_t nvtxRangeDesc = {};
nvtxRangeDesc.size = 64;
nvtxRangeDesc.ptr = ptr;

nvtxMemHeapDesc_t nvtxHeapDesc = {};
nvtxHeapDesc.extCompatID = NVTX_EXT_COMPATID_MEM;
nvtxHeapDesc.structSize = sizeof(nvtxMemHeapDesc_t);
nvtxHeapDesc.usage = NVTX_MEM_HEAP_USAGE_TYPE_SUB_ALLOCATOR;
nvtxHeapDesc.type = NVTX_MEM_TYPE_VIRTUAL_ADDRESS;
nvtxHeapDesc.typeSpecificDescSize = sizeof(nvtxMemVirtualRangeDesc_t);
nvtxHeapDesc.typeSpecificDesc = &nvtxRangeDesc;

auto nvtxPool = nvtxMemHeapRegister(
    nvtxDomain,
    &nvtxHeapDesc);
```

Please note that Compute Sanitizer only supports **nvtxMemHeapRegister** with parameters **usage = NVTX_MEM_HEAP_USAGE_TYPE_SUB_ALLOCATOR** and **type = NVTX_MEM_TYPE_VIRTUAL_ADDRESS**. If you are using the CUDA runtime API, **nvtxMemHeapRegister** can be used with allocations created with **cuMemAlloc**.

An existing pool can be reset to its initial state using **nvtxMemHeapReset**. The following example resets the pool previously allocated.

```
// Requires <nvtx3/nvToolsExtMem.h>

nvtxMemHeapReset(nvtxDomain, nvtxPool);
```

In a similar fashion, a pool can be unregistered using **nvtxMemHeapUnregister**. An allocation cannot be used after it is unregistered, but the allocation must be freed using **cudaFree** to dispose of it.

```
// Requires <nvtx3/nvToolsExtMem.h>

nvtxMemHeapUnregister(nvtxDomain, nvtxPool);
```

For your convenience, calling **cudaFree** on a memory pool causes Compute Sanitizer to automatically unregister it.

2.3.2. Suballocations Management

Once a pool is created, users can create suballocations within this pool using **nvtxMemRegionsRegister**. For your convenience, you can register multiple regions at

the same time. The following example creates a suballocation of 16 bytes at address **ptr**. Both **ptr** and **ptr + 16 bytes** must be part of the pool.

```
// Requires <nvtx3/nvToolsExtMem.h>

nvtxMemVirtualRangeDesc_t nvtxRangeDesc = {};
nvtxRangeDesc.size = 16;
nvtxRangeDesc.ptr = ptr;

nvtxMemRegionsRegisterBatch_t nvtxRegionsDesc = {};
nvtxRegionsDesc.extCompatID = NVTX_EXT_COMPATID_MEM;
nvtxRegionsDesc.structSize = sizeof(nvtxMemRegionsRegisterBatch_t);
nvtxRegionsDesc.regionType = NVTX_MEM_TYPE_VIRTUAL_ADDRESS;
nvtxRegionsDesc.heap = nvtxPool;
nvtxRegionsDesc.regionCount = 1;
nvtxRegionsDesc.regionDescElementSize = sizeof(nvtxMemVirtualRangeDesc_t);
nvtxRegionsDesc.regionDescElements = &nvtxRangeDesc;

nvtxMemRegionsRegister(nvtxDomain, &nvtxRegionsDesc);
```

For your convenience, Initcheck assumes that a new suballocation is uninitialized, meaning failure to initialize it might result in error reports. Please note that Compute Sanitizer only supports **nvtxMemRegionsRegister** with parameter **regionType = NVTX_MEM_TYPE_VIRTUAL_ADDRESS**. Suballocations are considered as regular allocations for NVTX naming and permissions API, therefore it is possible to label them or change their permissions.

Existing suballocations can be resized using **nvtxMemRegionsResize**. The following example resizes our previous suballocation at address **ptr** from 16 bytes to 32.

```
// Requires <nvtx3/nvToolsExtMem.h>

nvtxMemVirtualRangeDesc_t nvtxRangeDesc = {};
nvtxRangeDesc.size = 32;
nvtxRangeDesc.ptr = ptr;

nvtxMemRegionsResizeBatch_t nvtxRegionsDesc = {};
nvtxRegionsDesc.extCompatID = NVTX_EXT_COMPATID_MEM;
nvtxRegionsDesc.structSize = sizeof(nvtxMemRegionsResizeBatch_t);
nvtxRegionsDesc.regionType = NVTX_MEM_TYPE_VIRTUAL_ADDRESS;
nvtxRegionsDesc.regionDescCount = 1;
nvtxRegionsDesc.regionDescElementSize = sizeof(nvtxMemVirtualRangeDesc_t);
nvtxRegionsDesc.regionDescElements = &nvtxRangeDesc;

nvtxMemRegionsResize(nvtxDomain, &nvtxRegionsDesc);
```

In a similar fashion, existing allocations can be removed using **nvtxMemRegionsUnregister**. The following example removes our previous suballocation at address **ptr**.

```
nvtxMemRegionRef_t nvtxRegionRef;
nvtxRegionRef.pointer = ptr;

nvtxMemRegionsUnregisterBatch_t nvtxRegionsDesc = {};
nvtxRegionsDesc.extCompatID = NVTX_EXT_COMPATID_MEM;
nvtxRegionsDesc.structSize = sizeof(nvtxMemRegionsUnregisterBatch_t);
nvtxRegionsDesc.refType = NVTX_MEM_REGION_REF_TYPE_POINTER;
nvtxRegionsDesc.refCount = 1;
nvtxRegionsDesc.refElementSize = sizeof(nvtxMemRegionRef_t);
nvtxRegionsDesc.refElements = &nvtxRegionRef;

nvtxMemRegionsUnregister(nvtxDomain, &nvtxRegionsDesc);
```

Omitting to unregister a suballocation is reported as a memory leak if Compute Sanitizer is used in combination with option **`--leak-check yes`**.

2.4. Naming API

Any allocation can be assigned a name, so future Compute Sanitizer error reports can refer to an allocation by its name. This example names the allocation at address **`ptr`**: "My Allocation".

```
// Requires <nvtx3/nvToolsExtMem.h>

nvtxMemRegionNameDesc_t nvtxLabelDesc;
nvtxLabelDesc.regionRefType = NVTX_MEM_REGION_REF_TYPE_POINTER;
nvtxLabelDesc.nameType = NVTX_MESSAGE_TYPE_ASCII;
nvtxLabelDesc.region.pointer = ptr;
nvtxLabelDesc.name.ascii = "My Allocation";

nvtxMemRegionsNameBatch_t nvtxRegionsDesc = {};
nvtxRegionsDesc.extCompatID = NVTX_EXT_COMPATID_MEM;
nvtxRegionsDesc.structSize = sizeof(nvtxMemRegionsNameBatch_t);
nvtxRegionsDesc.regionCount = 1;
nvtxRegionsDesc.regionElementSize = sizeof(nvtxMemRegionNameDesc_t);
nvtxRegionsDesc.regionElements = &nvtxLabelDesc;

nvtxMemRegionsName(nvtxDomain, &nvtxRegionsDesc);
```

Please note that Compute Sanitizer only supports **`nvtxMemRegionsName`** with parameter **`nameType = NVTX_MESSAGE_TYPE_ASCII`** for all region elements in **`regionElements`**. As of now, only leak and unused memory reporting features allocation names.

2.5. Permissions API

2.5.1. Basic Permissions Management

NVTX Permissions API allows any allocation permissions to be restricted using **`nvtxMemPermissionsAssign`**. For this example, we use the global program scope (by calling **`nvtxMemCudaGetProcessWidePermissions`**), meaning permissions are applied

on all kernel launches. This example restricts the allocation at address **ptr** to read-only permissions.

```
// Requires <nvtx3/nvToolsExtMem.h> and <nvtx3/nvToolsExtMemCudaRt.h>

auto processPermHandle = nvtxMemCudaGetProcessWidePermissions(nvtxDomain);

nvtxMemPermissionsAssignRegionDesc_t nvtxPermDesc;
nvtxPermDesc.flags = NVTX_MEM_PERMISSIONS_REGION_FLAGS_READ;
nvtxPermDesc.regionRefType = NVTX_MEM_REGION_REF_TYPE_POINTER;
nvtxPermDesc.region.pointer = ptr;

nvtxMemPermissionsAssignBatch_t nvtxRegionsDesc = {};
nvtxRegionsDesc.extCompatID = NVTX_EXT_COMPATID_MEM;
nvtxRegionsDesc.structSize = sizeof(nvtxMemPermissionsAssignBatch_t);
nvtxRegionsDesc.permissions = processPermHandle;
nvtxRegionsDesc.regionCount = 1;
nvtxRegionsDesc.regionElementSize
 = sizeof(nvtxMemPermissionsAssignRegionDesc_t);
nvtxRegionsDesc.regionElements = &nvtxPermDesc;

nvtxMemPermissionsAssign(nvtxDomain, &nvtxRegionsDesc);
```

Valid permissions are:

- Read: **NVTX_MEM_PERMISSIONS_REGION_FLAGS_READ**
- Write: **NVTX_MEM_PERMISSIONS_REGION_FLAGS_WRITE**
- Atomic: **NVTX_MEM_PERMISSIONS_REGION_FLAGS_ATOMIC**
- A combination of read, write and atomic (using XORs).
- Reset: **NVTX_MEM_PERMISSIONS_REGION_FLAGS_RESET**

Using special permission **NVTX_MEM_PERMISSIONS_REGION_FLAGS_RESET** resets assigned permissions for the specified allocation on the specified scope.

Allocations permissions can be restricted on a per-device basis, using **nvtxMemCudaGetDeviceWidePermissions**. The following example gets the permissions handle from device **device**, a handle that is used with **nvtxMemPermissionsAssign** to change permissions for the allocation at address **ptr**, previously restricted to read-only on the global scope, and now read-write for kernel launched on **device** (no atomic allowed).

```
// Requires <nvtx3/nvToolsExtMem.h> and <nvtx3/nvToolsExtMemCudaRt.h>

auto devicePermHandle = nvtxMemCudaGetDeviceWidePermissions(nvtxDomain, device);

nvtxMemPermissionsAssignRegionDesc_t nvtxPermDesc;
nvtxPermDesc.flags = NVTX_MEM_PERMISSIONS_REGION_FLAGS_READ |
 NVTX_MEM_PERMISSIONS_REGION_FLAGS_WRITE;
nvtxPermDesc.regionRefType = NVTX_MEM_REGION_REF_TYPE_POINTER;
nvtxPermDesc.region.pointer = ptr;

nvtxMemPermissionsAssignBatch_t nvtxRegionsDesc = {};
nvtxRegionsDesc.extCompatID = NVTX_EXT_COMPATID_MEM;
nvtxRegionsDesc.structSize = sizeof(nvtxMemPermissionsAssignBatch_t);
nvtxRegionsDesc.permissions = devicePermHandle;
nvtxRegionsDesc.regionCount = 1;
nvtxRegionsDesc.regionElementSize
 = sizeof(nvtxMemPermissionsAssignRegionDesc_t);
nvtxRegionsDesc.regionElements = &nvtxPermDesc;

nvtxMemPermissionsAssign(nvtxDomain, &nvtxRegionsDesc);
```

2.5.2. Advanced Permissions Management

Permissions can be assigned to a specific stream scope thanks to custom permissions objects. You can create one using **nvtxMemPermissionsCreate**, and bind it to a scope using **nvtxMemPermissionsBind**. The following example restricts the allocation at address **ptr** to read-only permissions.

```
// Requires <nvtx3/nvToolsExtMem.h> and <nvtx3/nvToolsExtMemCudaRt.h>

// Create new permissions object.
auto permHandle = nvtxMemPermissionsCreate(nvtxDomain,
 NVTX_MEM_PERMISSIONS_CREATE_FLAGS_NONE);

nvtxMemPermissionsAssignRegionDesc_t nvtxPermDesc;
nvtxPermDesc.flags = NVTX_MEM_PERMISSIONS_REGION_FLAGS_READ;
nvtxPermDesc.regionRefType = NVTX_MEM_REGION_REF_TYPE_POINTER;
nvtxPermDesc.region.pointer = ptr;

nvtxMemPermissionsAssignBatch_t nvtxRegionsDesc = {};
nvtxRegionsDesc.extCompatID = NVTX_EXT_COMPATID_MEM;
nvtxRegionsDesc.structSize = sizeof(nvtxMemPermissionsAssignBatch_t);
nvtxRegionsDesc.permissions = permHandle;
nvtxRegionsDesc.regionCount = 1;
nvtxRegionsDesc.regionElementSize
 = sizeof(nvtxMemPermissionsAssignRegionDesc_t);
nvtxRegionsDesc.regionElements = &nvtxPermDesc;

// Assign read-only permissions to allocation at address ptr.
// Permissions will be applied on scope bound to permHandle.
nvtxMemPermissionsAssign(nvtxDomain, &nvtxRegionsDesc);

// Binding will happen on next kernel launch on this CPU thread, meaning the
// stream for this launch will be the one bound to this permissions object.
nvtxMemPermissionsBind(
    nvtxDomain,
    permHandle,
    NVTX_MEM_PERMISSIONS_BIND_SCOPE_CUDA_STREAM,
    NVTX_MEM_PERMISSIONS_BIND_FLAGS_NONE);

// permHandle is now bound to stream.
MyKernel<<<BlocksNb, ThreadsNb, 0, stream>>>(ptr);
```

On permissions object creation or binding, you can specify inheritance restriction flags. For example, excluding write permissions will block access for all allocations with unassigned permissions on that scope. These are applied:

- **nvtxMemPermissionsCreate**: applied for kernel launches on stream bound to the created object.
- **nvtxMemPermissionsBind**: applied for next kernel launch (on this CPU thread) and others using the same stream.

Please note that Compute Sanitizer only supports **nvtxMemPermissionsBind** with parameter **scope = NVTX_MEM_PERMISSIONS_BIND_SCOPE_CUDA_STREAM**.

Permissions objects currently bound can be unbound using **nvtxMemPermissionsUnbind** and destroyed using **nvtxMemPermissionsDestroy**. Permissions object destruction will result in an unbinding.

```
// Requires <nvtx3/nvToolsExtMem.h>

nvtxMemPermissionsUnbind(nvtxDomain,
 NVTX_MEM_PERMISSIONS_BIND_SCOPE_CUDA_STREAM)

nvtxMemPermissionsDestroy(nvtxDomain, permHandle);
```

Please note that Compute Sanitizer only supports **nvtxMemPermissionsUnbind** with parameter **scope = NVTX_MEM_PERMISSIONS_BIND_SCOPE_CUDA_STREAM**.

Peer devices access can be restricted for all allocations using **nvtxMemCudaSetPeerAccess**. If no permissions are set on an active scope for an allocation using **nvtxMemPermissionsAssign**, then default permissions set using **nvtxMemCudaSetPeerAccess** are applied. The following example restricts accesses to read-only on all devices except **device**.

```
// Requires <nvtx3/nvToolsExtMem.h>

auto permHandle = nvtxMemCudaGetDeviceWidePermissions(nvtxDomain, device);
nvtxMemCudaSetPeerAccess(
    nvtxDomain,
    permHandle,
    NVTX_MEM_CUDA_PEER_ALL_DEVICES,
    NVTX_MEM_PERMISSIONS_REGION_FLAGS_READ);
nvtxMemCudaSetPeerAccess(
    nvtxDomain,
    permHandle,
    device,
    NVTX_MEM_PERMISSIONS_REGION_FLAGS_READ |
 NVTX_MEM_PERMISSIONS_REGION_FLAGS_WRITE |
 NVTX_MEM_PERMISSIONS_REGION_FLAGS_ATOMIC);
```

Chapter 3. LIMITATIONS

Please note the Compute Sanitizer support for NVTX Memory API has the following limitations:

- Allocation names are visible on leak and unused memory reports, but not on other error reports for now.
- Allocation names must be encoded in ASCII, contain only printable characters, and contain between 1 and 49 characters (must comply to the following regex: `^[:print:]{1,49}$`)
- Permissions are only applied to kernel launches. Other operations, such as **`cudaMemcpy`** or **`cudaMemset`**, are not supported for now.

Notice

Trademarks

Copyright