COMPUTE SANITIZER

v2024.1.1 | March 2024

Compute Sanitizer

TABLE OF CONTENTS

LIST OF TABLES

Chapter 1.
INTRODUCTION

1.1. About Compute Sanitizer

Compute Sanitizer is a functional correctness checking suite included in the CUDA toolkit. This suite contains multiple tools that can perform different type of checks. The *memcheck* tool is capable of precisely detecting and attributing out of bounds and misaligned memory access errors in CUDA applications. The tool can also report hardware exceptions encountered by the GPU. The *racecheck* tool can report shared memory data access hazards that can cause data races. The *initcheck* tool can report cases where the GPU performs uninitialized accesses to global memory. The *synccheck* tool can report cases where the application is attempting invalid usages of synchronization primitives. This document describes the usage of these tools.

1.2. Why Compute Sanitizer

NVIDIA allows developers to easily harness the power of GPUs to solve problems in parallel using CUDA. CUDA applications often run thousands of threads in parallel. Every programmer invariably encounters memory access errors and thread ordering, hazards that are hard to detect and time consuming to debug. The number of such errors increases substantially when dealing with thousands of threads. The Compute Sanitizer suite is designed to detect those problems in your CUDA application.

1.3. How to Get Compute Sanitizer

Compute Sanitizer is installed as part of the CUDA toolkit.

1.4. Compute Sanitizer Tools

Compute Sanitizer provides different checking mechanisms through different tools. Currently the supported tools are:

‣ *Memcheck* – The memory access error and leak detection tool. See Memcheck Tool
‣ *Racecheck* – The shared memory data access hazard detection tool. See Racecheck Tool
‣ *Initcheck* – The uninitialized device global memory access detection tool. See Initcheck Tool
‣ *Synccheck* – The thread synchronization hazard detection tool. See Synccheck Tool

Chapter 2.
COMPUTE SANITIZER

Compute Sanitizer tools can be invoked by running the **compute-sanitizer** executable as follows:

```
compute-sanitizer [options] app_name [app_options]
```

For a full list of options that can be specified to compute-sanitizer and their default values, see Command Line Options

2.1. Command Line Options

Command line options can be specified to **compute-sanitizer**. With some exceptions, the options are usually of the form **--option value**. The option list can be terminated by specifying **--**. All subsequent words are treated as the application being run and its arguments.

The table below describes the supported options in detail. The first column is the option name passed to **compute-sanitizer**. Some options have a one character short form, which is given in parentheses. These options can be invoked using a single hyphen. For example, the help option can be invoked as **-h**. The options that have a short form do not take a value.

The second column contains the permissible values for the option. In case the value is user defined, it is shown below in braces **{}**. An option that can accept any numerical value is represented as *{number}*.

The third column contains the default value of the option. Some options have different default values depending on the architecture they are being run on.

Table 1 Compute Sanitizer command line options

Option	Values	Default	Description
check-device-heap	yes,no	yes	Enables checking of device heap allocations. This applies to both error checking and leak checking.

Option	Values	Default	Description
check-exit-code	yes, no	yes	Checks the application exit code and print an error if it is different than 0.
check-optix-leaks	N/A	disabled	Detects and reports OptiX resources that were created and were not destroyed at OptixDeviceContextDestroy time. For more information, see OptiX support.
check-warpgroup-mma	yes,no	yes	Enables memcheck and synccheck support for PTX `wgmma` instructions (requires sm_90a). For memcheck, the tool checks that the matrices loaded by `wgmma.mma_async` are in shared memory range. For synccheck, see Synccheck support for wgmma.
coredump-behavior	full,exit	full	Controls the behavior of the target application after generating a CUDA coredump. ▸ **full:** Abort the target application and generate a CPU coredump. ▸ **exit:** Exit the target application without generating a CPU coredump.
coredump-name	{filename}	N/A	Name to use for the generated coredump file.
demangle	full, simple, no	full	Enables the demangling of device function names. For more information, see Name Demangling.
destroy-on-device-error	context,kernel	context	This controls how the application proceeds on hitting a memory access error. For more information, see Error Actions.
error-exitcode	{number}	0	The exit code Compute Sanitizer will return if the original application succeeded but the tool detected that errors were present. This is meant to allow Compute Sanitizer to be integrated into automated test suites.
force-blocking-launches	N/A	disabled	This forces all host kernel launches to be sequential. When enabled, the number and precision of reported errors will decrease.
force-synchronization-limit	{number}	0	This forces a synchronization after a stream reaches the given number of launches without synchronizing. This is meant to reduce the memory usage of the Compute Sanitizer tools, but it can affect performances.
generate-coredump	N/A	disabled	When this is set, a coredump will be generated for the first error encountered and program execution will

Option	Values	Default	Description
			be stopped. For more information, see Coredump support.
help (h)	N/A	N/A	Displays the help message
injection-path	N/A	N/A	Sets the path to injection libraries.
injection-path32	N/A	N/A	Sets the path to 32bit injection libraries.
kernel-name	{key1=val1} [{,key2=val2}]	N/A	Controls which application kernels will be checked by the running the Compute Sanitizer tool. For more information, see Specifying Filters.
kernel-name-exclude	{key1=val1} [{,key2=val2}]	N/A	Controls which application kernels will be checked by the running the Compute Sanitizer tool. For more information, see Specifying Filters.
language	c,fortran	c	This controls the application source language specific behavior in Compute Sanitizer tools. For fortran specific behavior, see CUDA Fortran Specific Behavior.
c,launch-count	{number}	0	Limit the number of kernel launches to check. The count is only incremented for launches that match the kernel filters. Use 0 for unlimited.
s,launch-skip	{number}	0	Set the number of kernel launches to skip before starting to check. The count is only incremented for launches that match the kernel filters.
launch-timeout	{number}	10 for single process, 60 for multi-process	Timeout in seconds for the connection to the target process. A value of zero forces compute-sanitizer to wait infinitely.
log-file	{filename}	N/A	This is the file Compute Sanitizer will write all of its text output to. By default, Compute Sanitizer will print all output to stdout. For more information, see Escape Sequences.
max-connections	{number}	10	Maximum number of ports for connecting to the target application.
kill	N/A	disabled	Makes the compute-sanitizer kill the target application when a communication error is met. By default, the compute-sanitizer will instead await for the normal completion of the program without reporting potential errors.
mode	launch-and-attach,launch,attach	launch-and-attach	Select the mode of interaction with the target application

Option	Values	Default	Description
			▸ **launch-and-attach:** Launch the target application and immediately attach. ▸ **launch:** Launch the target application and suspend it, waiting for tool to attach. ▸ **attach:** Attach to a previously launched application to which no other tool is attached.
num-callers-device	{number}	0	Set the number of callers to print in device stack traces. Use 0 for unlimited.
num-callers-host	{number}	0	Set the number of callers to print in host stack traces. Use 0 for unlimited.
num-cuda-barriers	{number}	0	Set the number of cuda::barriers that the target application will use per block. Use 0 for automatic detection.
nvtx	true,false	true	Enable NVTX support.
port	{number}	49152	Base port for connecting to the target application.
prefix	{string}	========	The string prepended to Compute Sanitizer output lines.
print-level	info,warn,error,fatal	warn	The minimum print level of messages from Compute Sanitizer.
print-limit	{number}	100	When this option is set, Compute Sanitizer will stop printing errors after reaching the given number of errors. Use 0 for unlimited printing.
print-session-details	N/A	disabled	Print details about the sanitizer session for each target application such as process ID, command line, target system etc.
quiet,q	N/A	disabled	Controls whether to run silently and only print error messages.
read	{filename}	N/A	The input Compute Sanitizer file to read data from. This can be used in conjunction with the --save option to allow processing records after a run.
require-cuda-init	yes, no	yes	Controls whether Compute Sanitizer should return an error if the target application does not use CUDA.
save	{filename}	N/A	Filename where Compute Sanitizer will save the output from the current run. For more information, see Escape Sequences.
save-session-details	N/A	disabled	Save details about the sanitizer session for each target application in the file specified by `--save`. This option has no effect if the `--save` option is not used.

Option	Values	Default	Description
show-backtrace	yes,host,device,no	yes	Displays a backtrace for most types of errors. "no" disables all backtraces, "yes" enables all backtraces. "host" enables only host side backtraces. "device" enables only device side backtraces. For more information, see Stack Backtraces.
support-32bit	N/A	disabled	This option only exists on Linux x86_64. Enables the support for tracking application that includes 32-bit processes. On Windows, the support is always enabled if the 32bit injection libraries are found. Note: Only the 64bit processes are supported for actual checking, the purpose of the option is to allow tracking of the children process of a 32bit process.
suppressions	{filename}	N/A	Input XML file containing a list of reports that should be filtered out by the tool if detected. For more information, see Error suppression.
target-processes	application-only,all	all	Select which processes are to be tracked by compute-sanitizer: The root application process, or the root application and all its child processes.
target-processes-filter	{string}	N/A	Set the comma separated expressions to filter which processes are tracked. ▸ `<process name>` Set the process name to filter by. Only exactly matched processes are tracked. ▸ `regex:<expression>` Set the regex to filter matching process name profiling. On shells that recognize regular expression symbols as special characters (e.g. Linux bash), the expression needs to be escaped with quotes, e.g. `--target-processes-filter regex:".*Process"`. The executable name will be considered as process name to match. If the process name or the provided expression match, the process will be tracked. **Examples** `--target-processes-filter MatrixMul` Filter all processes having executable name exactly as "MatrixMul". `--target-processes-filter regex:Matrix` Filter all processes that include the string "Matrix" in their executable name, e.g. "MatrixMul" and "MatrixAdd".

Option	Values	Default	Description
			`--target-processes-filter MatrixMul,MatrixAdd`Filter all processes having executable name exactly as "MatrixMul" or "MatrixAdd".
tool	memcheck, racecheck, initcheck, synccheck	memcheck	Controls which Compute Sanitizer tool is actively running.
version (V)	N/A	N/A	Prints the version of Compute Sanitizer.
xml	N/A	disabled	Emit error output to file in XML format. When used, --save must also be set to specify the file to save to.

Table 2   *Memcheck* tool command line options

Option	Values	Default	Description
check-cache-control	N/A	disabled	Check cache control memory accesses.
detect-missing-module-unload	N/A	disabled	Detect leaks caused by missing module unload calls. This option can report false positives if the application uses the CUDA runtime as it depends on the destruction order between runtime and driver when the application exits which is not guaranteed.
ignore-getprocaddress-notfound	N/A	disabled	Ignore CUDA_ERROR_NOT_FOUND API errors for cuGetProcAddress.
leak-check	full,no	no	Prints information about all allocations that have not been freed via cudaFree at the point when the context was destroyed. For more information, see Leak Checking.
padding	{number}	0	Makes the compute-sanitizer allocate padding buffers after every CUDA allocation. *number* is the size in bytes of a padding buffer. Fore more information, see Padding.
report-api-errors	all, explicit, no	explicit	Reports errors if any CUDA API call fails. For more information, see CUDA API Error Checking.
track-stream-ordered-races arg	all,use-before-alloc,use-after-free,no	no	Track CUDA stream-ordered allocations races. For more information, see Stream-ordered race detection.

Table 3   *Racecheck* tool command line options

Option	Values	Default	Description
racecheck-detect-level	{info,warn,error}	warn	Set the minimum level of race conditions to detect.
racecheck-indirect-barrier-dependency	N/A	disabled	Enables tracking of indirect cuda::barrier dependency in racecheck. Avoids false positives when the target application is relying on chains of arrive-waits on multiple different barriers to synchronize shared memory accesses between threads that would not have participated in the same barrier. Using this option may have a performance impact.
racecheck-memcpy-async	yes,no	yes	Enables check for asynchronous memory copy operations. For more information, see Racecheck support for asynchronous copy.
racecheck-num-workers	{number}	0	Number of CPU worker threads used by the tool. Use 0 for automatic.
racecheck-report	hazard,analysis,all	analysis	Controls how racecheck reports information. For more information, see Racecheck Report Modes.

Table 4   *Initcheck* tool command line options

Option	Values	Default	Description
check-api-memory-access	yes,no	yes	Enables checking of cudaMemcpy/cudaMemset
check-optix	N/A	disabled	Check OptiX kernel launches with initcheck. For more information, see OptiX support.
track-unused-memory	N/A	disabled	Check for unused memory allocations.
unused-memory-threshold	{number}	0	Threshold in percentage under which unused memory reports are silenced. The value needs to be a number between 0 and 100.

Table 5   *Synccheck* tool command line options

Option	Values	Default	Description
missing-barrier-init-is-fatal	yes,no	yes	Controls whether a missing `cuda::barrier` initialization will exit the warp.

2.2. Compilation Options

The Compute Sanitizer tools do not need any special compilation flags to function.

The output displayed by the Compute Sanitizer tools is more useful with some extra compiler flags. The **-G** option to nvcc forces the compiler to generate debug information for the CUDA application. To generate line number information for applications without affecting the optimization level of the output, the **-lineinfo** nvcc option can be used. The Compute Sanitizer tools fully support both of these options and can display source attribution of errors for applications compiled with line information.

The stack backtrace feature of the Compute Sanitizer tools is more useful when the application contains function symbol names. For the host backtrace, this varies based on the host OS. On Linux, the host compiler must be given the **-rdynamic** option to retain function symbols. On Windows, the application must be compiled for debugging, i.e. the **/Zi** option. When using nvcc, flags to the host compiler can be specified using the **-Xcompiler** option. For the device backtrace, the full frame information is only available when the application is compiled with device debug information. The compiler can skip generation of frame information when building with optimizations.

Sample command line to build with function symbols and device side line information on Linux:

```
nvcc -Xcompiler -rdynamic -lineinfo  -o out in.cu
```

2.3. Environment Variables

The following environment variables can be set before launching the compute-sanitizer tool.

Table 6   Environment Variables

Name	Description	Default/Values
NV_COMPUTE_SANITIZER_BINARY_PATCHING	Controls whether compute-sanitizer will instrument user kernel code. This option is intended for debugging and should not be used by normal users.	Default if unset: enabled. Valid values: any positive value between 0 and INT_MAX.
NV_COMPUTE_SANITIZER_LOCAL_CONNECTION_OVERRIDE	Override the default local connection mechanism between frontend and target processes. The default mechanism is platform-dependent. This should only be used if there are connection problems between	Default: unset (use default mechanism) Set to "uds" to use Unix Domain Socket connections (available on Posix platforms, only). Set to "tcp" to use TCP (available on all platforms). Set to "named-pipes"

Name	Description	Default/Values
	frontend and target processes in a local launch.	to use Windows Named Pipes (available on Windows, only).
NV_COMPUTE_SANITIZER_MAX_RACECHECK_CLUSTER_RECORDS	Override the maximum number of racecheck cluster access records for early exit race detection. This option can be used to either increase the number of races the tool can detect, or to suppress early exit races (0 will display no early exit race).	Default if unset: 100. Valid values: any positive value between 0 and `INT_MAX`.
NV_COMPUTE_SANITIZER_MAX_RACECHECK_HAZARDS	Override the maximum number of racecheck hazards tool will process. This option can be used to either increase the number of races the tool can detect, or to reduce it and save host memory.	Default if unset: 10,000,000. Valid values: any positive value between 0 and `INT_MAX`.
NV_COMPUTE_SANITIZER_SHARED_ADDRESSING_SUPPORT	Override shared memory addressing support.	Default if unset: `auto`. Set to `none` to disable shared addressing support. Set to `force` to force shared addressing support. Set to `auto` to enable shared memory addressing support if system supports HMM or ATS.

Chapter 3.
MEMCHECK TOOL

3.1. What is Memcheck?

The *memcheck* tool is a run time error detection tool for CUDA applications. The tool can precisely detect and report out of bounds and misaligned memory accesses to global, local and shared memory in CUDA applications. It can also detect and report hardware reported error information. In addition, the memcheck tool can detect and report memory leaks in the user application.

3.2. Supported Error Detection

The errors that can be reported by the memcheck tool are summarized in the table below. The location column indicates whether the report originates from the host or from the device. The precision of an error is explained in the paragraph below.

Table 7 Memcheck reported error types

Name	Description	Location	Precision	See also
Memory access error	Errors due to out of bounds or misaligned accesses to memory by a global, local, shared or global atomic access.	Device	Precise	
Hardware exception	Errors that are reported by the hardware error reporting mechanism.	Device	Imprecise	
Malloc/Free errors	Errors that occur due to incorrect use of `malloc()/free()` in CUDA kernels.	Device	Precise	Device Side Allocation Checking
CUDA API errors	Reported when a CUDA API call in the application returns a failure.	Host	Precise	CUDA API Error Checking
cudaMalloc memory leaks	Allocations of device memory using `cudaMalloc()` that	Host	Precise	Leak Checking

Name	Description	Location	Precision	See also
	have not been freed by the application.			
Device Heap Memory Leaks	Allocations of device memory using `malloc()` in device code that have not been freed by the application.	Device	Imprecise	Device Side Allocation Checking

The memcheck tool reports two classes of errors *precise* and *imprecise*.

Precise errors in memcheck are those that the tool can uniquely identify and gather all information for. For these errors, memcheck can report the block and thread coordinates of the thread causing the failure, the program counter (PC) of the instruction performing the access, as well as the address being accessed and its size and type. If the CUDA application contains line number information (by either being compiled with device side debugging information, or with line information), then the tool will also print the source file and line number of the erroneous access.

Imprecise errors are errors reported by the hardware error reporting mechanism that could not be precisely attributed to a particular thread. The precision of the error varies based on the type of the error and in many cases, memcheck may not be able to attribute the cause of the error back to the source file and line.

3.3. Using Memcheck

The memcheck tool is enabled by default when running the Compute Sanitizer application. It can also be explicitly enabled by using the `--tool memcheck` option.

```
compute-sanitizer --tool memcheck [sanitizer_options] app_name [app_options]
```

When run in this way, the memcheck tool will look for precise, imprecise, malloc/free and CUDA API errors. The reporting of device leaks must be explicitly enabled. Errors identified by the memcheck tool are displayed on the screen after the application has completed execution. See Understanding Memcheck Errors for more information about how to interpret the messages printed by the tool.

3.4. Understanding Memcheck Errors

The memcheck tool can produce a variety of different errors. This is a short guide showing some samples of errors and explaining how the information in each error report can be interpreted.

1. *Memory access error*: Memory access errors are generated for errors that the memcheck tool can correctly attribute and identify the erroneous instruction. Below is an example of a precise memory access error.

```
========= Invalid __global__ write of size 4 bytes
=========     at unaligned_kernel():0x160 in memcheck_demo.cu:6
=========     by thread (0,0,0) in block (0,0,0)
=========     Address 0x7f6510c00001 is misaligned
```

Let us examine this error line by line:

```
Invalid __global__ write of size 4 bytes
```

The first line shows the memory segment, type and size being accessed. The memory segment is one of:

- ▶ __global__ : for device global memory
- ▶ __shared__ : for per block shared memory
- ▶ __local__ : for per thread local memory

In this case, the access was to device global memory. The next field contains information about the type of access, whether it was a read or a write. In this case, the access is a write. Finally, the last item is the size of the access in bytes. In this example, the access was 4 bytes in size.

```
at unaligned_kernel():0x160 in memcheck_demo.cu:6
```

The second line contains the CUDA kernel name, offset of the instruction, the source file and line number (if available). In this example, the instruction causing the access was at offset 0x160 inside the **unaligned_kernel** CUDA kernel. Additionally, since the application was compiled with line number information, this instruction corresponds to line 6 in the memcheck_demo.cu source file.

```
by thread (0,0,0) in block (0,0,0)
```

The third line contains the thread indices and block indices of the thread on which the error was hit. In this example, the thread doing the erroneous access belonged to the first thread in the first block.

```
Address 0x7f6510c00001 is misaligned
```

The fourth line contains the memory address being accessed and the type of access error. The type of access error can either be out of bounds access or misaligned access. In this example, the access was to address 0x7f6510c00001 and the access error was because this address was not aligned correctly.

2. *Hardware exception*: Imprecise errors are generated for errors that the hardware reports to the memcheck tool. Hardware exceptions have a variety of formats and messages. Typically, the first line will provide some information about the type of error encountered.

3. *Malloc/free error*: Malloc/free errors refer to the errors in the invocation of device side **malloc()/free()** in CUDA kernels. An example of a malloc/free error:

```
========= Malloc/Free error encountered : Double free
=========     at 0x79d8
=========     by thread (0,0,0) in block (0,0,0)
=========     Address 0x400aff920
```

We can examine this line by line.

```
Malloc/Free error encountered : Double free
```

The first line indicates that this is a malloc/free error, and contains the type of error. This type can be:

- ▸ Double free – This indicates that the thread called **free()** on an allocation that has already been freed.
- ▸ Invalid pointer to free – This indicates that **free** was called on a pointer that was not returned by **malloc()**.
- ▸ Heap corruption : This indicates generalized heap corruption, or cases where the state of the heap was modified in a way that memcheck did not expect.

In this example, the error is due to calling **free()** on a pointer which had already been freed.

```
at 0x79d8
```

The second line gives the PC on GPU where the error was reported. This PC is usually inside of system code, and is not interesting to the user. The device frame backtrace will contain the location in user code where the **malloc()/free()** call was made.

```
by thread (0,0,0) in block (0,0,0)
```

The third line contains the thread and block indices of the thread that caused this error. In this example, the thread has threadIdx = (0,0,0) and blockIdx = (0,0,0)

```
Address 0x400aff920
```

This line contains the value of the pointer passed to **free()** or returned by **malloc()**

4. *Leak errors*: Errors are reported for allocations created using cudaMalloc and for allocations on the device heap that were not freed when the CUDA context was destroyed. An example of a cudaMalloc allocation leak report is the following:

```
========= Leaked 64 bytes at 0x400200200
```

The error message reports information about the size of the allocation that was leaked as well as the address of the allocation on the device.

A device heap leak message will be explicitly identified as such:

```
========= Leaked 16 bytes at 0x4012ffff6 on the device heap
```

5. *CUDA API error*: CUDA API errors are reported for CUDA API calls that return an error value. An example of a CUDA API error:

```
========= Program hit invalid copy direction for memcpy (error 21) on CUDA
 API call to cudaMemcpy.
```

The message contains the returned value of the CUDA API call, as well as the name of the API function that was called.

3.5. CUDA API Error Checking

The memcheck tool supports reporting an error if a CUDA API call made by the user program returned an error. The tool supports this detection for both CUDA run time and CUDA driver API calls. In all cases, if the API function call has a nonzero return value, Compute Sanitizer will print an error message containing the name of the API call that failed and the return value of the API call.

CUDA API error reports do not terminate the application, they merely provide extra information. It is up to the application to check the return status of CUDA API calls and handle error conditions appropriately.

The following API errors are not reported:

▶ **cudaErrorNotReady** for **cudaEventQuery** and **cudaStreamQuery** APIs.
▶ **cudaErrorPeerAccessAlreadyEnabled** for **cudaDeviceEnablePeerAccess** API.
▶ **cudaErrorPeerAccessNotEnabled** for **cudaDeviceDisablePeerAccess** API.

3.6. Device Side Allocation Checking

The *memcheck* tool checks accesses to allocations in the device heap.

These allocations are created by calling **malloc()** inside a kernel. This feature is implicitly enabled and can be disabled by specifying the **--check-device-heap no** option. This feature is only activated for kernels in the application that call **malloc()**.

The tool will report an error if the application calls a **free()** twice for the same allocation, or if it calls **free()** on an invalid pointer.

> Make sure to look at the device side backtrace to find the location in the application where the **malloc()/free()** call was made.

3.7. Leak Checking

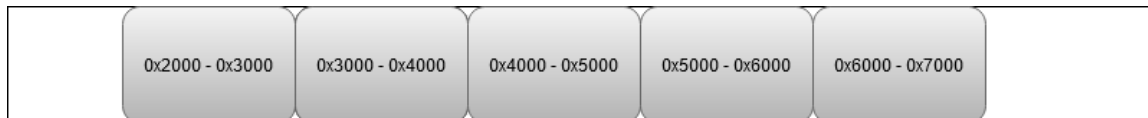The *memcheck* tool can detect leaks of allocated memory.

Memory leaks are device side allocations that have not been freed by the time the context is destroyed. The *memcheck* tool tracks device memory allocations created using the CUDA driver or runtime APIs.

The **--leak-check full** option must be specified to enable leak checking.
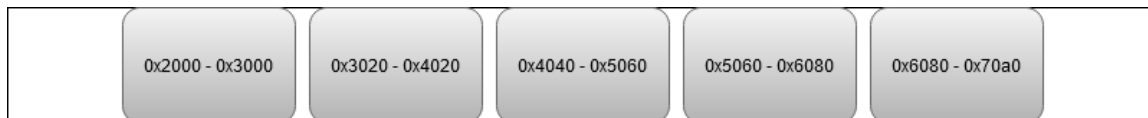
3.8. Padding

The *memcheck* tool can automatically add padding to memory allocations in order to improve out of bounds error detection for global memory.

By default, global memory buffers can be allocated back-to-back in the virtual address space. When that happens, an overflow access into the first buffer will simply happen in the second buffer and not be detected as out-of-bounds.

0x2000 - 0x3000	0x3000 - 0x4000	0x4000 - 0x5000	0x5000 - 0x6000	0x6000 - 0x7000

Using the **--padding** option will automatically extend the allocation size, effectively creating a padding buffer after each allocation. This improves the out of bounds error detection as accesses to the padding area will always be considered invalid. The example below displays possible buffer addresses when using **--padding 32**. Every allocation is followed by a 32 bytes padding buffer. Writing or reading this buffer will cause an out-of-bounds access to be reported.

0x2000 - 0x3000	0x3020 - 0x4020	0x4040 - 0x5060	0x5060 - 0x6080	0x6080 - 0x70a0

This option supports allocations created via the **cudaMalloc** APIs, **cudaHostAlloc** and **cudaMallocHost**.

This option does not support allocations created via **cudaHostRegister** or the CUDA virtual memory management APIs.

Be aware that using this option will result in an increased device memory pressure, potentially causing additional CUDA out of memory errors.

3.9. Stream-ordered race detection
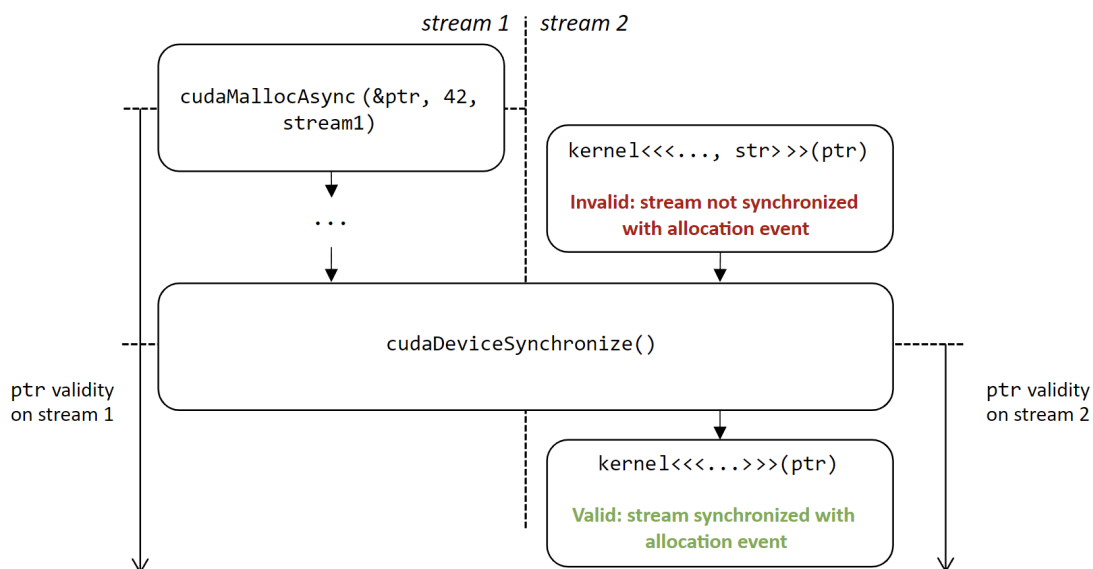
The *memcheck* tool can detect stream-ordered allocations races using the **--track-stream-ordered-races all** option. It will report accesses to stream-ordered allocations used outside of their lifespan.

The tool is capable of detecting 2 types of races:

▸ Use-before-alloc races (**--track-stream-ordered-races use-before-alloc**)

This race occurs when an allocation is used before it is available: an allocation created using **cudaMallocAsync** on a stream cannot be used on another stream without a prior synchronization event after the allocation.

It also includes cases where an allocation is freed before it is available using **cudaFreeAsync**.

▶ Use-after-free races (`--track-stream-ordered-races use-after-free`)

This race occurs when an allocation is used after it is freed: an allocation freed using **cudaFreeAsync** on a stream cannot be used on another stream without a following synchronization event before the free.

Chapter 4.
RACECHECK TOOL

4.1. What is Racecheck?

The *racecheck* tool is a run time shared memory data access hazard detector. The primary use of this tool is to help identify memory access race conditions in CUDA applications that use shared memory.

In CUDA applications, storage declared with the **__shared__** qualifier is placed on chip *shared memory*. All threads in a thread block can access this per block shared memory. Shared memory goes out of scope when the thread block completes execution. As shared memory is on chip, it is frequently used for inter-thread communication and as a temporary buffer to hold data being processed. As this data is being accessed by multiple threads in parallel, incorrect program assumptions may result in data races. Racecheck is a tool built to identify these hazards and help users write programs free of shared memory races.
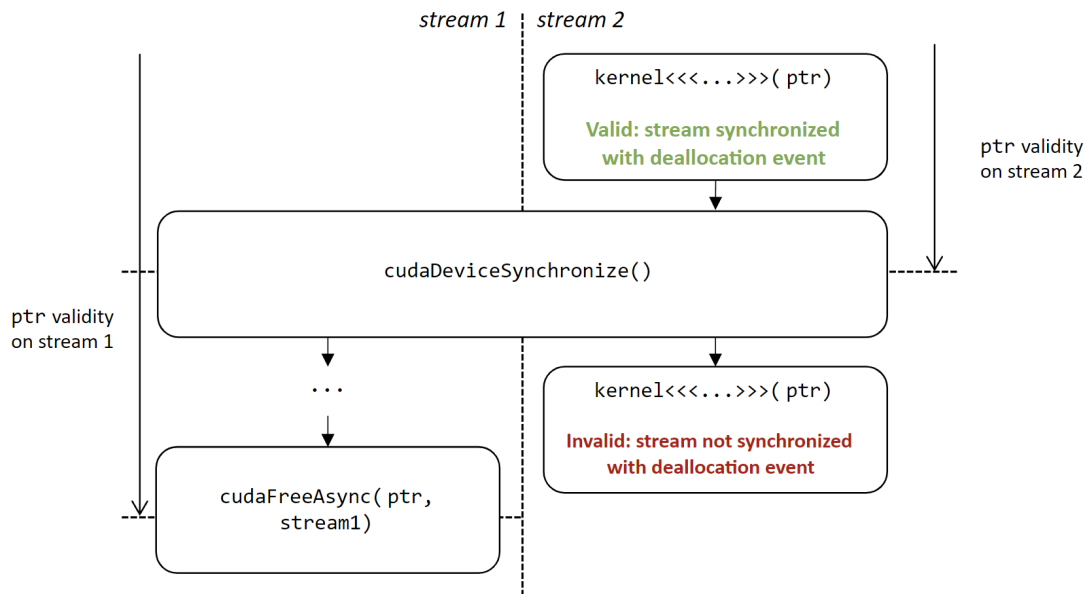
Currently, this tool only supports detecting accesses to on-chip shared memory.

4.2. What are Hazards?

A *data access hazard* is a case where two threads attempt to access the same location in memory resulting in non-deterministic behavior, based on the relative order of the two accesses. These hazards cause *data races* where the behavior or the output of the application depends on the order in which all parallel threads are executed by the hardware. Race conditions manifest as intermittent application failures or as failures when attempting to run a working application on a different GPU.

The racecheck tool identifies three types of canonical hazards in a program. These are :

▸   Write-After-Write (*WAW*) hazards

   This hazard occurs when two threads attempt to write data to the same memory location. The resulting value in that location depends on the relative order of the two accesses.

▶ Write-After-Read (*WAR*) hazards

This hazard occurs when two threads access the same memory location, with one thread performing a read and another a write. In this case, the writing thread is ordered before the reading thread and the value returned to the reading thread is not the original value at the memory location.

▶ Read-After-Write (*RAW*) hazards

This hazard occurs when two threads access the same memory location, with one thread performing a read and the other a write. In this case, the reading thread reads the value before the writing thread commits it.

4.3. Using Racecheck

The racecheck tool is enabled by running the Compute Sanitizer application with the `--tool racecheck` option.

```
compute-sanitizer --tool racecheck [sanitizer_options] app_name [app_options]
```

Once racecheck has identified a hazard, the user can make program modifications to ensure this hazard is no longer present. In the case of Write-After-Write hazards, the program should be modified so that multiple writes are not happening to the same location. In the case of Read-After-Write and Write-After-Read hazards, the reading and writing locations should be deterministically ordered. In CUDA kernels, this can be achieved by inserting a `__syncthreads()` call between the two accesses. To avoid races between threads within a single warp, `__syncwarp()` can be used.

> 💬 The racecheck tool does not perform any memory access error checking. It is recommended that users first run the memcheck tool to ensure the application is free of errors.

4.4. Racecheck Report Modes

The racecheck tool can produce two types of output:

▶ *Hazard* reports

These reports contain detailed information about one particular hazard. Each hazard report is byte accurate and represents information about conflicting accesses between two threads that affect this byte of shared memory.

▶ *Analysis* reports

These reports contain a post analysis set of reports. These reports are produced by the racecheck tool by analysing multiple hazard reports and examining active device state. For example usage of analysis reports, see Understanding Racecheck Analysis Reports.

4.5. Understanding Racecheck Analysis Reports

In *analysis* reports, the racecheck tool produces a series of high-level messages that identify the source locations of a particular race, based on observed hazards and other machine state.

A sample racecheck analysis report is below:

```
========= WARNING: Race reported between Write access at RAW()+0xf0 in
 raceGroupBasic.cu:40
=========        and Read access at RAW()+0x280 in raceGroupBasic:46 [4 hazards]
```

The analysis record contains high-level information about the hazard that is conveyed to the end user. Each line contains information about a unique location in the application which is participating in the race.

The first word on the first line indicates the severity of this report. In this case, the message is at the WARNING level of severity. For more information on the different severity levels, see Racecheck Severity Levels. Analysis reports are composed of one or more racecheck hazards, and the severity level of the report is that of the hazard with the highest severity.

The first line additionally contains the type of access. The access can be either:

▸ Read
▸ Write

The next item on the line is the name of the kernel issuing the access and the offset of the location where the access happened from. In this case, the offset is 0xf0 in the RAW() kernel. If the application was compiled with line number information, this line also contains the file name and line number of the access.

The next lines contain the location of the other offsets participating in the race condition. In this case, there is only one other location which is the RAW() kernel at offset 0x280. Similarly to the first line, file name and line number are printed if the application was compiled with line number information. Finally, the line also contains the number of hazards detected for this specific race condition.

A given analysis report will always contain at least one line which is performing a write access. A common strategy to eliminate races which contain only write accesses is to ensure that the write access is performed by only one thread. In the case of races with multiple readers and one writer, introducing explicit program ordering via a `__syncthreads()` call can avoid the race condition. For races between threads within the same warp, the `__syncwarp()` intrinsic can be used to avoid the hazard.

4.6. Understanding Racecheck Hazard Reports

In *hazard* reporting mode, the racecheck tool produces a series of messages detailing information about hazards in the application. The tool is byte accurate and produces a

message for each byte on which a hazard was detected. Additionally, when enabled, the host backtrace for the launch of the kernel will also be displayed.

A sample racecheck hazard is below:

```
========= ERROR: Potential WAW hazard detected at __shared__ 0x0 in block
 (0,0,0) :
=========        Write Thread (0,0,0) at WAW()+0x2f0 in raceWAW.cu:20
=========        Write Thread (1,0,0) at WAW()+0x2f0 in raceWAW.cu:20
=========        Current Value : 1, Incoming Value : 2
```

The hazard records are dense and capture a lot of interesting information. In general terms, the first line contains information about the hazard severity, type and address, as well as information about the thread block where it occurred. The next 2 lines contain detailed information about the two threads that were in contention. These two lines are ordered chronologically, so the first entry is for the access that occurred earlier and the second for the access that occurred later. The final line is printed for some hazard types and captures the actual data that was being written.

Examining this line by line, we have :

```
ERROR: Potential WAW hazard detected at __shared__ 0x0 in block (0, 0, 0)
```

The first word on this line indicates the severity of this hazard. In this case, the message is at the ERROR level of severity. For more information on the different severity levels, see Racecheck Severity Levels.

The next piece of information here is the type of hazard. The racecheck tool detects three types of hazards:

- ▸ WAW or Write-After-Write hazards
- ▸ WAR or Write-After-Read hazards
- ▸ RAW or Read-After-Write hazards

The type of hazard indicates the accesses types of the two threads that were in contention. In this example, the hazard is of Write-After-Write type.

The next piece of information is the address in shared memory that was being accessed. This is the offset in per block shared memory that was being accessed by both threads. Since the racecheck tool is byte accurate, the message is only for the byte of memory at given address. In this example, the byte being accessed is byte 0x0 in shared memory.

Finally, the first line contains the block index of the thread block to which the two racing threads belong.

The second line contains information about the first thread to write to this location.

```
Write Thread (0, 0, 0) at WAW()+0x2f0 in raceWAW.cu:20(void)
```

The first item on this line indicates the type of access being performed by this thread to the shared memory address. In this example, the thread was writing to the location. The next component is the index of the thread block. In this case, the thread is at index (0,0,0). Following this, we have the name of the kernel and byte offset of the instruction which did the access in the kernel. In this example, the offset is 0x2f0. This is followed by the source file and line number (if line number information is available).

The third line contains similar information about the second thread that was causing this hazard. This line has an identical format to the previous line.

The fourth line contains information about the data in the two accesses.

```
Current Value : 1, Incoming Value : 2
```

If the second thread in the hazard was performing a write access, i.e., the hazard is a Write-After-Write (WAW) or a Write-After-Read (WAR), this line contains the value after the access by the first thread as the *Current Value* and the value that will be written by the second access as the *Incoming Value*. In this case, the first thread wrote the value 1 to the shared memory location. The second thread is attempting to write the value 2.

4.7. Racecheck Severity Levels

Problems reported by racecheck can be of different severity levels. Depending on the level, different actions are required from developers. By default, only issues of severity level WARNING and ERROR are shown. The command line option **--print-level** can be used to set the lowest severity level that should be reported.

Racecheck reports have one of the following severity levels:

▸ *INFO*: The lowest level of severity. This is for hazards that have no impact on program execution and hence are not contributing to data access hazards. It is still a good idea to find and eliminate such hazards.

▸ *WARNING*: Hazards at this level of severity are determined to be programming model hazards, however may be intentionally created by the programmer.
An example of this are hazards due to warp level programming that make the assumption that threads are proceeding in groups. Such hazards are typically only encountered by advanced programmers. In cases where a beginner programmer encounters such errors, he should treat them as sources of hazards.

Starting with the Volta architecture, programmers cannot rely anymore on the assumption that threads within a warp execute in lock-step unconditionally.
As a result, warnings due to warp-synchronous programming without explicit synchronization must be fixed when developing or porting applications from earlier architectures to Volta and above. Developers can use the **__syncwarp()** intrinsic or the Cooperative Groups API.

▸ *ERROR*: The highest level of severity. This corresponds to hazards that are very likely candidates for causing data access races. Programmers would be well advised to examine errors at this level of severity.

4.8. Racecheck support for `cuda::barrier`

Racecheck supports synchronization through **cuda::barrier** on Ampere GPUs and newer.

The number of barriers tracked by the tool is based on the number of barriers present in the source code as reported by compiler information. In some cases, the compiler may

undercount this number. Racecheck will report the following warning if more barriers are used than expected:

```
========= Warning: Detected overflow of tracked cuda::barrier structures.
 Results might be incorrect. Try using --num-cuda-barriers to fix the issue
```

The **--num-cuda-barriers** option can be used to indicate the number of expected barriers in the source code and workaround this issue.

4.9. Racecheck support for asynchronous copy

Racecheck supports race detection on shared memory for asynchronous memory copy operations from global to shared memory introduced in compute capability 8.0. These can take the form of CUDA C++ **cuda::memcpy_async** or the PTX **cp.async**. Specifically, racecheck is able to detect when the target of a asynchronous copy tracked by a pipeline (CUDA C++) or async-group (PTX) was accessed before the required commit/wait to guarantee its completion. In these cases, individual hazards when using **--racecheck-report hazard** will bear the mention **(invalid memcpy_async synchronization)**. These checks can be disabled by using **--racecheck-memcpy-async no**.

4.10. Racecheck cluster entry and exit race detection

Racecheck supports race detection on remote shared memory accesses without appropriate cluster-wide synchronization. When a kernel makes a remote shared memory access from one block to another (in the same cluster), it needs to guarantee that the target block exists, otherwise error **cudaErrorLaunchFailure** is raised. One way to achieve this is using cluster.sync() from the Cluster Group API . Refer to the CUDA documentation about distributed shared memory for more information.

When running a program under Racecheck, instead of failing, the tool will report these two types of illegal accesses:

1. Late entry race detection: a block is trying to access shared memory from another block in the cluster without an appropriate cluster-wide synchronization beforehand.
2. Early exit race detection: a block is trying to access shared memory from another block in the cluster without an appropriate cluster-wide synchronization before the target block exits.

A sample report for both races is below:

```
========= Potential invalid __shared__ read of size 4 bytes
=========       at RemoteAccess(int *, int)+0x170 in RaceCluster.cu:10
=========       by thread (0,0,0) in block (0,0,0)
=========       Address 0x1000400 is located in a block that might not have
 entered yet
=========
========= Potential invalid __shared__ read of size 4 bytes
=========       at RemoteAccess(int *, int)+0x170 in RaceCluster.cu:10
=========       by thread (0,0,0) in block (0,0,0)
=========       Address 0x1000400 is located in a block that might have already
 exited
=========
```

Chapter 5.
INITCHECK TOOL

5.1. What is Initcheck?

The *initcheck* tool is a run time uninitialized device global memory access detector. This tool can identify when device global memory is accessed without it being initialized via device side writes, or via CUDA memcpy and memset API calls.

Currently, this tool only supports detecting accesses to device global memory.

5.2. Using Initcheck

The initcheck tool is enabled by running the Compute Sanitizer application with the **--tool initcheck** option.

```
compute-sanitizer --tool initcheck [sanitizer_options] app_name [app_options]
```

> The initcheck tool does not perform any memory access error checking. It is recommended that users first run the memcheck tool to ensure the application is free of errors.

5.3. Unused memory detection

The initcheck tool can also be used to detect unused memory by using the **--track-unused-memory** option.

```
compute-sanitizer --tool initcheck --track-unused-memory yes app_name
 [app_options]
```

A sample unused memory report is below:

```
=========   Unused memory in allocation 0x7fed9f400000 of size 100 bytes
=========      Not written 80 bytes at offset 0x14 (0x7fed9f400014)
=========      80% of allocation were unused.
```

This report contains the address and size of the allocation, the number of bytes not used and their location. The location can be a range if all unused bytes are not contiguous.

The behavior for this feature can be adjusted with the **--unused-memory-threshold** option which takes the minimum percentage at which reports should be printed. For instance, using a value of 81 or above would silence the sample report above.

Chapter 6.
SYNCCHECK TOOL

6.1. What is Synccheck?

The *synccheck* tool is a runtime tool that can identify whether a CUDA application is correctly using synchronization primitives, specifically `__syncthreads()` and `__syncwarp()` intrinsics and their Cooperative Groups API counterparts.

6.2. Using Synccheck

The synccheck tool is enabled by running the Compute Sanitizer application with the `--tool synccheck` option.

```
compute-sanitizer --tool synccheck [sanitizer_options] app_name [app_options]
```

> The synccheck tool does not perform any memory access error checking. It is recommended that users first run the memcheck tool to ensure the application is free of errors.

6.3. Understanding Synccheck Reports

For each violation, the synccheck tool produces a report message that identifies the source location of the violation and its classification.

A sample synccheck report is below:

```
========= Barrier error detected. Divergent thread(s) in warp
=========      at ThreadDivergence(int *, int)+0xf0 in divergence.cu:79
=========      by thread (37,0,0) in block (0,0,0)
```

Each report starts with "Barrier error detected." In most cases, this is followed by a classification of the detected barrier error. In this message, a CUDA block with divergent threads was found. The following error classes can be reported:

▸ *Divergent thread(s) in block*: Divergence between threads within a block was detected for a barrier that does not support this on the current architecture. For example, this occurs when `__syncthreads()` is used within conditional code but the conditional does not evaluate equally across all threads in the block.

▸ *Divergent thread(s) in warp*: Divergence between threads within a single warp was detected for a barrier that does not support this on the current architecture.

▸ *Invalid arguments*: A barrier instruction or primitive was used with invalid arguments. This can occur for example if not all threads reaching a `__syncwarp()` declare themselves in the mask parameter. However, synccheck will not detect cases where not all the threads declared in the mask parameter reach the `__syncwarp()`.

The next line states the offset within the function of the location where the access happened. In this case, the offset is 0xf0. If the application was compiled with line number information, this line would also contain the file name and line number of the access, followed by the name of the kernel issuing the access.

The third line contains information on the thread and block for which this violation was detected. In this case, it is thread 37 in block 0.

6.4. Synccheck support for `cuda::barrier`

Synccheck supports synchronization through `cuda::barrier` on Ampere GPUs and newer.

The number of barriers tracked by the tool is based on the number of barriers present in the source code as reported by compiler information. In some cases, the compiler may undercount this number. Synccheck will report the following warning if more barriers are used than expected:

```
========= Warning: Detected overflow of tracked cuda::barrier structures.
 Results might be incorrect. Try using --num-cuda-barriers to fix the issue
```

The `--num-cuda-barriers` option can be used to indicate the number of expected barriers in the source code and workaround this issue.

6.5. Synccheck support for `wgmma`

Synccheck supports additional checks related to PTX `wgmma` instructions for Hopper sm_90a architecture.

`wgmma` instructions are executed across a warpgroup. Each warp in the warpgroup are expected to execute the same `wgmma` instructions in the same order with the same predicates, with all threads active or none. Synccheck can detect and report cases where these rules are not respected, and will exit the entire warpgroup when detected. In such cases, the report will start with "Warpgroup MMA sequence error detected" instead of

"Barrier error detected", followed by a description of the specific error encountered. The error is reported once per warp encountering the error.

The **`--check-warpgroup-mma`** option can be used to enable or disable these checks.

Chapter 7.
COMPUTE SANITIZER FEATURES

7.1. Nonblocking Mode

By default, the standalone Compute Sanitizer tool will launch kernels in nonblocking mode. This allows the tool to support error reporting in applications running concurrent kernels

To force kernels to execute serially, a user can use the **`--force-blocking-launches yes`** option. One side effect is that when in blocking mode, only the first thread to hit an error in a kernel will be reported. Also, using this option or **`--force-synchronization-limit`** will disable CUDA reduced API serialization.

7.2. Stack Backtraces

Compute Sanitizer can generate backtraces when given **`--show-backtrace`** option. Backtraces usually consist of two sections – a saved host backtrace that leads up to the CUDA driver call site, and a device backtrace at the time of the error. Each backtrace contains a list of frames showing the state of the stack at the time the backtrace was created.

To get function names in the host backtraces, the user application must be built with support for symbol information in the host application. For more information, see Compilation Options

Backtraces are printed for most Compute Sanitizer tool outputs, and the information generated varies depending on the type of output. The table below explains the kind of host and device backtrace seen under different conditions.

Table 8 Compute Sanitizer Stack Backtrace Information

Output Type	Host Backtrace	Device Backtrace
Memory access error	Kernel launch on host	Precise backtrace on device

Output Type	Host Backtrace	Device Backtrace
Hardware exception	Kernel launch on host	Imprecise backtrace on device [1]
Malloc/Free error	Kernel launch on host	Precise backtrace on device
cudaMalloc allocation leak	Callsite of cudaMalloc	N/A
CUDA API error	Callsite of CUDA API call	N/A
Compute Sanitizer internal error	Callsite leading to internal error	N/A
Device heap allocation leak	N/A	N/A
Shared memory hazard	Kernel launch on host	N/A

Note that for OptiX applications, the name of OptiX internal device functions will be displayed as "NVIDIA Internal".

7.3. Name Demangling

The Compute Sanitizer suite supports displaying mangled and demangled names for CUDA kernels and CUDA device functions. By default, tools display the fully demangled name, which contains the name of the kernel as well as its prototype information. In the simple demangle mode, the tools will only display the first part of the name. If demangling is disabled, tools will display the complete mangled name of the kernel.

7.4. Dynamic Parallelism

The Compute Sanitizer tool suite supports dynamic parallelism. The *memcheck* tool supports precise error reporting of out of bounds and misaligned accesses on global, local and shared memory accesses, as well as on global atomic instructions for applications using dynamic parallelism. In addition, the imprecise hardware exception reporting mechanism is also fully supported. Error detection on applications using dynamic parallelism requires significantly more memory on the device; as a result, in memory constrained environments, *memcheck* may fail to initialize with an internal out of memory error.

For limitations, see the known limitations in the Release Notes section.

7.5. Error Actions

When encountering an error, Compute Sanitizer behavior depends on the type of error. The default behavior of Compute Sanitizer is to continue execution on purely host side errors. Hardware exceptions detected by the memcheck tool cause the CUDA context to

---

[1] In some cases, there may be no device backtrace

be destroyed. Precise errors (such as memory access and malloc/free errors) detected by the memcheck tool cause the kernel to be terminated. This terminates the kernel without running any subsequent instructions and the application continues launching other kernels in the CUDA context. The handling of memory access and malloc/free errors detected by the memcheck tool can be changed using the **--destroy-on-device-error** option.

The **--destroy-on-device-error kernel** option is not supported on Maxwell GPUs.

For racecheck detected hazards, the hazard is reported, but execution is not affected.

For a full summary of error action, based on the type of the error see the table below. The error action *terminate kernel* refers to the cases where the kernel is terminated early, and no subsequent instructions are run. In such cases, the CUDA context is not destroyed and other kernels continue execution and CUDA API calls can still be made.

> When kernel execution is terminated early, the application may not have completed its computations on data. Any subsequent kernels that depend on this data will have undefined behavior.

The action *terminate CUDA context* refers to the cases where the CUDA context is forcibly terminated. In such cases, all outstanding work for the context is terminated and subsequent CUDA API calls will fail. The action *continue application* refers to cases where the application execution is not impacted, and the kernel continues executing instructions.

Table 9 Compute Sanitizer Error Actions

Error Type	Location	Action	Comments
Memory access error	Device	Terminate CUDA context	User can choose to instead terminate the kernel
Hardware exception	Device	Terminate CUDA context	Subsequent calls on the CUDA context will fail
Malloc/Free error	Device	Terminate CUDA context	User can choose to instead terminate the kernel
cudaMalloc allocation leak	Host	Continue application	Error reported. No other action taken.
CUDA API error	Host	Continue application	Error reported. No other action taken.
Device heap allocation leak	Device	Continue application	Error reported. No other action taken.
Shared memory hazard	Device	Continue application	Error reported. No other action taken.
Synchronization error	Device	Terminate CUDA context	User can choose to instead terminate the kernel
Compute Sanitizer internal error	Host	Undefined	The application may behave in an undefined fashion

7.6. Escape Sequences

The **--save** and **--log-file** options to Compute Sanitizer accept the following escape sequences in the file name.

- ▸ **%%** : Replaced with a literal %.
- ▸ **%p** : Replaced with the PID of the Compute Sanitizer frontend application.
- ▸ **%q{ENVVAR}** : Replaced with the contents of the environment variable **ENVVAR**. If the variable does not exist, this is replaced with an empty string.
- ▸ Any other character following the % causes an error.

7.7. Specifying Filters

Compute Sanitizer tools support filtering the choice of kernels which should be checked. When a filter is specified, only kernels matching the filter will be checked. Filters are specified using the **--kernel-name** and **--kernel-name-exclude** options. By default, the Compute Sanitizer tools will check all kernels in the application.

The **--kernel-name** and **--kernel-name-exclude** options can be specified multiple times. If a kernel satisfies any filter, it will be checked by the running the Compute Sanitizer tool.

The **--kernel-name** and **--kernel-name-exclude** options take a filter specification consisting of a list of comma separated key value pairs, specified as **key=value**. When using the regex filter key, multiple key value pairs need to be specified through multiple use of the option instead. In order for a filter to be matched, all components of the filter specification must be satisfied. If a filter is incorrectly specified in any component, the entire filter is ignored. For a full summary of valid key values, see the table below. If a key has multiple strings, any of the strings can be used to specify that filter component.

Table 10   Compute Sanitizer Filter Keys

Name	Key String	Value	Comments
Kernel Name	kernel_name, kne	Complete mangled kernel name	User specifies the complete mangled kernel name.
Kernel Substring	kernel_substring, kns	Any substring in mangled kernel name	User specifies a substring in the mangled kernel name.
Regex	regex	Any regex that can be matched in a substring of the mangled kernel name	User specifies a regular expression searched in the mangled kernel name.

When using the **kernel-name** filters, the Compute Sanitizer tools will check all **device** function calls made by the kernel. When using CUDA Dynamic Parallelism (CDP), the Compute Sanitizer tools will not check child kernels launched from a checked kernel unless the child kernel matches a filter. If a GPU launched kernel that does not match a filter calls a device function that is reachable from a kernel that does match a filter, the

device function behaves as though it was checked. In the case of some tools, this can result in undefined behavior.

Filter usage example

We consider an application that launches three different kernels declared below.

```
__global__ void gamma(int *bufer);
__global__ void delta(int *bufer);
__global__ void epsilon(int *bufer);
```

Their respective mangled names are **_Z5gammaPi**, **_Z5deltaPi** and **_Z7epsilonPi**. We only want to check the launches of the kernel epsilon. Here are different means to achieve it:

- **compute-sanitizer --kernel-name kne=_Z7epsilonPi** Only epsilon is matching the specified filter, so only kernel launches of epsilon will be checked.
- **compute-sanitizer --kernel-name kns=epsilon** Since "epsilon" is a substring of "_Z7epsilonPi", and also happens to be the only kernel having this substring in its mangled name, only epsilon will be matched and checked.
- **compute-sanitizer --kernel-name-exclude kns=delta,kne=_Z5gammaPi** This time, we are using the exclude options. Only epsilon is not matched by the exclude option in this scenario, which means it will be the only one checked. We specified multiple filter separating them with a comma: this can be used with both **kernel-name** and **kernel-name-exclude**.
- **compute-sanitizer --kernel-name-exclude kns=delta --kernel-name-exclude kne=_Z5gammaPi** Same as above, except we used the exclude option twice to specify multiple filters instead of specifying them all at once. If needed, **kernel-name** and **kernel-name-exclude** can be used at the same time.
- **compute-sanitizer --kernel-name regex='[a-z]{7}'** For this example we are using the regex filter. It matches any kernel for which the regular expression can be matched anywhere inside the mangle named. The specified regex matches any 7 consecutive lowercase letter. Only **_Z7epsilonPi** has 7 consecutive lowercase letter, and therefore is the only kernel matched by **--kernel-name**.

7.8. Coredump support

Starting from CUDA 11.6, the compute-sanitizer tool can generate a CUDA coredump once an error is detected by using the **--generate-coredump yes** option. Once the coredump is generated, the target application will abort.

The coredump file can be loaded in cuda-gdb using the following option:

```
(cuda-gdb) target cudacore core.name.nvcudmp
```

See the cuda-gdb documentation for more information.

The `--coredump-name` option can be used to specify the file name of the coredump. See the "Naming of GPU core dump files" section of the cuda-gdb documentation for more information on template specifiers and default name.

The coredump feature has the following restrictions:

▸ Only threads that encountered an error can be inspected in the generated coredump
▸ Maxwell GPUs are not supported
▸ The racecheck tool is not supported.
▸ Coredumps are not supported on WSL2.

7.9. Error suppression

The compute-sanitizer tools can sometimes generate false positive reports. In these cases, a suppression file can be provided as input to the tool to suppress the reporting of these false positives.

A suppression file can be generated by using the `--xml` option of the compute-sanitizer tool on the target application. Once generated, the XML file can be edited manually to be more generic.

On subsequent use of the tools, the suppression file can be provided as input using the `--suppressions` option.

The following rules are applied when checking if a detected report should be suppressed:

▸ The types of the report must match.
▸ If provided in the suppression file, integer fields must match exactly.
▸ If provided in the suppression file, a string field can be a regex.
▸ When comparing stack traces, the suppression trace needs to have the same number of frames or less than the report one.
▸ Stack frame comparisons include the following fields (if provided in the suppression): function name, file name and module name.

The following types of error can be suppressed:

▸ CUDA API errors
▸ Initcheck uninitialized memory accesses
▸ Racecheck analysis reports

7.10. OptiX support

Starting from CUDA 11.6, the compute-sanitizer tool support OptiX 7 applications with memcheck and initcheck. The option `--check-optix yes` needs to be set for optix launches to be tracked with initcheck. To get full device backtrace information, please make sure your OptiX modules are compiled with `OPTIX_COMPILE_DEBUG_LEVEL_FULL` set in the `debugLevel` field in the `OptixModuleCompileOptions` structure.

When using compute-sanitizer on OptiX applciations, it is possible that some or all device frames are located in OptiX internal code. Such frames have their name displayed as **NVIDIA Internal**. See the example below of an error reported in user code called from an internal OptiX function:

```
========= Invalid __global__ write of size 1 bytes
=========     at __raygen__placeholder_0x67b9a77bb7822a34+0x19b0 in /home/cuda/
optixApp.cu:70
=========     by thread (0,0,0) in block (0,0,0)
=========     Address 0x7f91edf00403 is out of bounds
=========     and is 262,132 bytes after the nearest allocation at
 0x7f91edec0400 of size 16 bytes
=========     Device Frame:NVIDIA Internal [0x520]
=========     Saved host backtrace up to driver entry point at kernel launch
 time
[...]
```

Starting from CUDA 11.7, it is possible to detect leaks of **OptixModule**, **optixPipeline**, **optixProgramGroup** and **optixDenoiser** with compute-sanitizer. This requires using the **--check-optix-leaks yes** option. Leaks will only reported if the **OptixDeviceContext** is destroyed with a call to **OptixDeviceContextDestroy**. **OptixDeviceContext** that are leaking will have their associated CUDA buffers reported with a regular use of **--leak-check full**. See the example below of an **optixProgramGroup** that was not destroyed being reported:

```
========= Leaked an OptixProgramGroup with handle 0x55dbffbd9840
=========     Saved host backtrace up to driver entry point at allocation time
[...]
```

The following feature set is supported per OptiX API version:

OptiX API Version	Kernel checks	Resource leak check
7.0 - 8.0	Yes	Yes

Chapter 8.
USAGE GUIDE

8.1. Memory Footprint

Compute Sanitizer tools can have a large memory footprint due to their tracking data. This can cause out of memory errors on applications performing a large number of concurrent kernel launches.

```
========= Internal Sanitizer Error: The Sanitizer encountered an error while
 launching kernel_name and didn't track the launch. Errors might go undetected.
 (Unable to allocate enough memory to perform the requested operation)
```

The tools might also cause a failure to allocate host memory causing the application to crash.

```
========= Error: process didn't terminate successfully
========= Target application returned an error
```

This issue can be resolved using one of the following command line options:

- **`--force-synchronization-limit {number}`** forces a stream synchronization after a stream reaches the given number of launches without synchronizing.
- **`--force-blocking-launches yes`** forces the serialization of of every kernel launch. This option is equivalent to **`--force-synchronization-limit 1`**.

Using CUDA lazy module loading will also help lower the memory footprint of the tools, both for host and device memory.

Chapter 9.
OPERATING SYSTEM SPECIFIC BEHAVIOR

This section describes operating system specific behavior.

9.1. Windows Specific Behavior

▸ Timeout Detection and Recovery (TDR)

On Windows, GPUs have a timeout associated with them. GPU applications that take longer than the threshold (default of 2 seconds) will be killed by the operating system. Since the Compute Sanitizer tools increase the runtime of kernels, it is possible for a CUDA kernel to exceed the timeout and therefore be terminated due to the TDR mechanism.

For the purposes of debugging, the number of seconds before which the timeout is hit can be modified by setting the timeout value in seconds in the DWORD registry key **TdrDelay** at:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\GraphicsDrivers
```

More information about the registry keys to control the Timeout Detection and Recovery mechanism is available from MSDN at http://msdn.microsoft.com/en-us/library/windows/hardware/ff569918%28v=vs.85%29.aspx.

9.2. Using the Compute Sanitizer on Jetson and Tegra devices

By default, on Jetson and Drive Tegra devices, GPU debugging is supported only if **compute-sanitizer** is launched by a user who is a member of the **debug** group.

To add the current user to the **debug** group run this command:

```
sudo usermod -a -G debug $USER
```

Chapter 10.
CUDA FORTRAN SUPPORT

This section describes support for CUDA Fortran.

10.1. CUDA Fortran Specific Behavior

▸ By default, error reports printed by Compute Sanitizer contain 0-based C style values for thread index (threadIdx) and block index (blockIdx). For Compute Sanitizer tools to use Fortran style 1-based offsets, use the **--language fortran** option.
▸ The CUDA Fortran compiler may insert extra padding in shared memory. Accesses hitting this extra padding may not be reported as an error.

Chapter 11.
COMPUTE SANITIZER TOOL EXAMPLES

11.1. Example Use of Memcheck

This section presents a walk-through of running the memcheck tool from Compute Sanitizer on a simple application called **memcheck_demo**.

> 💬 Depending on the SM type of your GPU, your system output may vary.

The application can be found on the compute-sanitizer github repository

This application can be compiled using the provided Makefile:

```
make
```

11.1.1. `memcheck_demo` Output

When a CUDA application causes access violations, the kernel launch may report an illegal memory access or misaligned address. Sticky errors will be reported for all subsequent kernel launches.

This sample application is causing two failures but there is no way to detect where the misaligned address access is caused. The second kernel is also not able to run, as illustrated in the following output:

```
$ ./memcheck_demo
Mallocing memory
Running unaligned_kernel: misaligned address
Running out_of_bounds_kernel: misaligned address
```

11.1.2. `memcheck_demo` Output with Memcheck (Release Build)

In this case, since the application is built in release mode, the Compute Sanitizer output contains only the kernel names from the application causing the access

violation. Though the kernel name and error type are detected, there is no line number information on the failing kernel. Also included in the output are the host and device backtraces for the call sites where the functions were launched

Now run this application with Compute Sanitizer and check the output. By default, the application will run so that the kernel is terminated on memory access errors, but other work in the CUDA context can still proceed.

In the output below, the first kernel no longer reports an unspecified launch failure as its execution has been terminated early after Compute Sanitizer detected the error. The application continued to run the second kernel. The error detected in the second kernel causes it to terminate early.

```
$ make run_memcheck
/usr/local/cuda/compute-sanitizer/compute-sanitizer --destroy-on-device-error
 kernel memcheck_demo
========= COMPUTE-SANITIZER
Mallocing memory
========= Invalid __global__ write of size 4 bytes
=========        at unaligned_kernel()+0x70
=========        by thread (0,0,0) in block (0,0,0)
=========        Address 0x7f671ac00001 is misaligned
=========        and is inside the nearest allocation at 0x7fb654c00000 of size 4
 bytes
=========        Saved host backtrace up to driver entry point at kernel launch
 time
=========        Host Frame: [0x2774ec]
=========                    in /lib/x86_64-linux-gnu/libcuda.so.1
=========        Host Frame:__cudart803 [0xfccb]
=========                    in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
=========        Host Frame:cudaLaunchKernel [0x6a578]
=========                    in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
=========        Host Frame:cudaError cudaLaunchKernel<char>(char const*, dim3,
 dim3, void**, unsigned long, CUstream_st*) [0xb535]
=========                    in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
=========        Host Frame:__device_stub__Z16unaligned_kernelv() [0xb22e]
=========                    in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
=========        Host Frame:unaligned_kernel() [0xb28c]
=========                    in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
=========        Host Frame:run_unaligned() [0xaf55]
=========                    in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
=========        Host Frame:main [0xb0e2]
=========                    in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
=========        Host Frame:../sysdeps/nptl/
libc_start_call_main.h:58:__libc_start_call_main [0x2dfd0]
=========                    in /lib/x86_64-linux-gnu/libc.so.6
=========        Host Frame:../csu/libc-start.c:379:__libc_start_main [0x2e07d]
=========                    in /lib/x86_64-linux-gnu/libc.so.6
=========        Host Frame:_start [0xada5]
=========                    in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
=========
Running unaligned_kernel: no error
========= Invalid __global__ write of size 4 bytes
=========        at out_of_bounds_kernel()+0x90
=========        by thread (0,0,0) in block (0,0,0)
=========        and is 140,418,624,437,472 bytes before the nearest allocation at
 0x7fb649a00000 of size 1,024 bytes
=========        Saved host backtrace up to driver entry point at kernel launch
 time
=========        Host Frame: [0x2774ec]
=========                    in /lib/x86_64-linux-gnu/libcuda.so.1
=========        Host Frame:__cudart803 [0xfccb]
=========                    in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
=========        Host Frame:cudaLaunchKernel [0x6a578]
=========                    in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
=========        Host Frame:cudaError cudaLaunchKernel<char>(char const*, dim3,
 dim3, void**, unsigned long, CUstream_st*) [0xb535]
=========                    in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
=========        Host Frame:__device_stub__Z20out_of_bounds_kernelv() [0xb34e]
=========                    in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
=========        Host Frame:out_of_bounds_kernel() [0xb3ac]
=========                    in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
=========        Host Frame:run_out_of_bounds() [0xb037]
=========                    in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
```

11.1.3. `memcheck_demo` Output with Memcheck (Debug Build)

The application can be built with device side debug information and function symbols as:

```
make dbg=1
```

The source location of the error is now reported in the compute-sanitizer output:

```
$ make run_memcheck
========= COMPUTE-SANITIZER
========= Invalid __global__ write of size 4 bytes
=========     at unaligned_kernel()+0x160 in /home/cuda/github/compute-
sanitizer-samples/Memcheck/memcheck_demo.cu:34
=========     by thread (0,0,0) in block (0,0,0)
=========     Address 0x7f3d7ce00001 is misaligned
=========     and is inside the nearest allocation at 0x7f9544c00000 of size 4
 bytes
=========     Saved host backtrace up to driver entry point at kernel launch
 time
=========     Host Frame: [0x2774ec]
=========                 in /lib/x86_64-linux-gnu/libcuda.so.1
=========     Host Frame:__cudart803 [0xfccb]
=========                 in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
=========     Host Frame:cudaLaunchKernel [0x6a578]
=========                 in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
=========     Host Frame:cudaError cudaLaunchKernel<char>(char const*, dim3,
 dim3, void**, unsigned long, CUstream_st*) [0xb535]
=========                 in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
=========     Host Frame:__device_stub__Z16unaligned_kernelv() [0xb22e]
=========                 in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
=========     Host Frame:unaligned_kernel() [0xb28c]
=========                 in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
=========     Host Frame:run_unaligned() [0xaf55]
=========                 in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
=========     Host Frame:main [0xb0e2]
=========                 in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
=========     Host Frame:../sysdeps/nptl/
libc_start_call_main.h:58:__libc_start_call_main [0x2dfd0]
=========                 in /lib/x86_64-linux-gnu/libc.so.6
=========     Host Frame:../csu/libc-start.c:379:__libc_start_main [0x2e07d]
=========                 in /lib/x86_64-linux-gnu/libc.so.6
=========     Host Frame:_start [0xada5]
=========                 in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
=========
Running unaligned_kernel: no error
========= Invalid __global__ write of size 4 bytes
=========     at out_of_bounds_function()+0xb0 in /home/cuda/github/compute-
sanitizer-samples/Memcheck/memcheck_demo.cu:39
=========     by thread (0,0,0) in block (0,0,0)
=========     Address 0x87654320 is out of bounds
=========     and is 140,276,689,190,112 bytes before the nearest allocation at
 0x7f953da00000 of size 1,024 bytes
=========     Device Frame:out_of_bounds_kernel()+0x30 in /home/cuda/github/
compute-sanitizer-samples/Memcheck/memcheck_demo.cu:44
=========     Saved host backtrace up to driver entry point at kernel launch
 time
=========     Host Frame: [0x2774ec]
=========                 in /lib/x86_64-linux-gnu/libcuda.so.1
=========     Host Frame:__cudart803 [0xfccb]
=========                 in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
=========     Host Frame:cudaLaunchKernel [0x6a578]
=========                 in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
=========     Host Frame:cudaError cudaLaunchKernel<char>(char const*, dim3,
 dim3, void**, unsigned long, CUstream_st*) [0xb535]
=========                 in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
=========     Host Frame:__device_stub__Z20out_of_bounds_kernelv() [0xb34e]
=========                 in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
=========     Host Frame:out_of_bounds_kernel() [0xb3ac]
=========                 in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
=========     Host Frame:run_out_of_bounds() [0xb037]
```

11.1.4. Leak Checking in Compute Sanitizer

To print information about the allocations that have not been freed at the time the CUDA context is destroyed, we can specify the **--leak-check full** option to Compute Sanitizer.

When running the program with the leak check option, the user is presented with a list of allocations that were not destroyed, along with the size of the allocation and the address on the device of the allocation. For allocations made on the host, each leak report will also print a backtrace corresponding to the saved host stack at the time the allocation was first made. Also presented is a summary of the total number of bytes leaked and the corresponding number of allocations.

In this example, the program created an allocation using **cudaMalloc()** and has not called **cudaFree()** to release it, leaking memory. Notice that Compute Sanitizer still prints errors it encountered while running the application. They are omitted in the output below for the sake of clarity.

```
$ make run_leakcheck
========= COMPUTE-SANITIZER
...
========= Leaked 1,024 bytes at 0x7fab4fa00000
=========       Saved host backtrace up to driver entry point at cudaMalloc time
=========       Host Frame: [0x9b5c16]
=========                   in /lib/x86_64-linux-gnu/libcuda.so.1
=========       Host Frame:__cudart612 [0x41f5e]
=========                   in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
=========       Host Frame:__cudart618 [0x1080b]
=========                   in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
=========       Host Frame:cudaMalloc [0x4f3ef]
=========                   in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
=========       Host Frame:main [0xb0dd]
=========                   in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
=========       Host Frame:../sysdeps/nptl/
libc_start_call_main.h:58:__libc_start_call_main [0x2dfd0]
=========                   in /lib/x86_64-linux-gnu/libc.so.6
=========       Host Frame:../csu/libc-start.c:379:__libc_start_main [0x2e07d]
=========                   in /lib/x86_64-linux-gnu/libc.so.6
=========       Host Frame:_start [0xada5]
=========                   in /home/cuda/github/compute-sanitizer-samples/
Memcheck/memcheck_demo
=========
========= LEAK SUMMARY: 1024 bytes leaked in 1 allocations
========= ERROR SUMMARY: 3 errors
```

11.2. Example Use of Racecheck

This section presents two example usages of the racecheck tool from Compute Sanitizer. The first example uses an application called **block_error**, which has shared memory

hazards on the block level. The second example uses an application called **warp_error**, which has shared memory hazards on the warp level.

> Depending on the SM type of your GPU, your system output may vary.

11.2.1. Block-level Hazards

The application can be found on the compute-sanitizer github repository

This application can be compiled using the provided Makefile:

```
make dbg=1
```

Each kernel thread write some element in shared memory. Afterward, thread 0 computes the sum of all elements in shared memory and stores the result in global memory variable **sum_out**.

Running this application under the racecheck tool with the **--racecheck-report analysis** option, the following error is reported:

```
$ make run_block_error
========= COMPUTE-SANITIZER
========= Error: Race reported between Write access at sumKernel(int *,
 int *)+0x90 in /home/cuda/github/compute-sanitizer-samples/Racecheck/
block_error.cu:41
=========      and Read access at sumKernel(int *, int *)+0x100 in /home/cuda/
github/compute-sanitizer-samples/Racecheck/block_error.cu:51 [508 hazards]
=========
========= RACECHECK SUMMARY: 1 hazard displayed (1 error, 0 warnings)
```

Racecheck reports races between thread 0 reading all shared memory elements in line 51 and each individual thread writing its shared memory entry in line 41. Accesses to shared memory between multiple threads, where at least one access is a write, can potentially race with each other. Since the races are between threads of different warps, the block-level synchronization barrier **__syncthreads()** is required in line 42.

Note that a total of 508 hazards are reported: the kernel uses a single block of 128 threads. The data size written or read, respectively, by each thread is four bytes (one **int**) and hazards are reported at the byte level. The writes by all threads race with the reads by thread 0, except for the four writes by thread 0 itself.

11.2.2. Warp-level Hazards

The application can be found on the compute-sanitizer github repository

This application can be compiled using the provided Makefile:

```
make dbg=1
```

The kernel computes the some of all individual elements in shared memory two stages. First, each thread computes its local shared memory value in **smem_first**. Second, a single thread of each warp is chosen with **if (tx % WARP_SIZE == 0)** to sum all elements written by its warp, indexed **wx**, and store the result in **smem_second**. Finally, thread 0 of the kernel computes the sum of elements in **smem_second** and writes the value into global memory.

Running this application under the racecheck tool with the **--racecheck-report hazard** option, multiple hazards with WARNING severity are reported:

```
========= Warning: (Warp Level Programming) Potential RAW hazard detected at
__shared__ 0x8c in block (0,0,0) :
=========        Write Thread (35,0,0) at sumKernel(int *, int *)+0x90 in /home/
cuda/github/compute-sanitizer-samples/Racecheck/warp_error.cu:44
=========        Read Thread (32,0,0) at sumKernel(int *, int *)+0x120 in /home/
cuda/github/compute-sanitizer-samples/Racecheck/warp_error.cu:56
=========        Current Value : 35
```

To avoid the errors demonstrated in the Block-level Hazards example, the kernel uses the block-level barrier **__syncthreads()** in line 60. However, racecheck still reports read-after-write (RAW) hazards between threads within the same warp, with severity WARNING. On architectures prior to SM 7.0 (Volta), programmers commonly relied on the assumption that threads within a warp execute code in lock-step (warp-level programming). Starting with CUDA 9.0, programmers can use the new **__syncwarp()** warp-wide barrier (instead of only **__syncthreads()** beforehand) to avoid such hazards. This barrier should be inserted at line 45.

11.3. Example Use of Initcheck

This section presents the usage of the initcheck tool from Compute Sanitizer. The example uses an application called **memset_error**.

11.3.1. Memset Error

The application can be found on the compute-sanitizer github repository

This application can be compiled using the provided Makefile:

```
make dbg=1
```

The example implements a very simple vector addition, where the thread index is added to each vector element. The vector contains **NumBlocks * NumThreads** elements of type **int**. The vector is allocated on the device and then initialized to 0 using **cudaMemset** before the kernel is launched.

Running this application under the initcheck tool reports multiple errors like the following:

```
$ make run_initcheck
========= Uninitialized __global__ memory read of size 4 bytes
=========     at vectorAdd(int *)+0x70 in /home/cuda/github/compute-sanitizer-
samples/Initcheck/memset_error.cu:41
=========     by thread (31,0,0) in block (1,0,0)
=========     Address 0x7f3c7ec000fc
```

The problem is that the call to **cudaMemset** expects the size of the to-be set memory in bytes. However, the size is given in elements, as a factor of **sizeof(int)** is missing while computing the parameter. As a result, 3/4 of the memory will have undefined values during the vector addition.

11.4. Example Use of Synccheck

This section presents two example usages of the synccheck tool from Compute Sanitizer. The first example uses an application called **divergent_threads**. The second example uses an application called **illegal_syncwarp**.

> Depending on the SM type of your GPU, your system output may vary.

11.4.1. Divergent Threads

The divergent_threads application can be found on the compute-sanitizer github repository

This application can be compiled using the provided Makefile:

```
make dbg=1
```

In this example, we launch a kernel with a single block of 64 threads. The kernels loops over **DataBlocks** blocks of input data **data_in**. In each iteration, **NumThreads** elements are added concurrently in shared memory. Finally, a single thread 0 computes the sum of all values in shared memory and writes it to **sum_out**.

Running this application under the synccheck tool, 16 errors like the following are reported:

```
$ make run_divergent_threads
========= Barrier error detected. Divergent thread(s) in warp
=========     at myKernel(int*, int*, int)+0x578 in divergent_thread.cu:54
=========     by thread (32,0,0) in block (0,0,0)
```

The issue is with the **__syncthreads()** in line 20 when reading the last data block into shared memory. Note that the last data block only has 48 elements (compared to 64

elements for all other blocks). As a result, not all threads of the second warp execute this statement in convergence as required.

> 💬 Calling `__syncthreads()` without convergence is allowed on SM 7.0 and above. Syncheck will not report any error for this example on these architectures.

11.4.2. Illegal Syncwarp

The illegal_syncwarp application can be found on the compute-sanitizer github repository

This application can be compiled using the provided Makefile:

```
make dbg=1
```

This example only applies to devices of compute capability 7.0 (Volta) and above. The kernel is launched with a single warp (32 threads), but only thread 0-15 are part of the computation. Each of these threads initializes one shared memory element with its thread index. After the assignment, `__syncwarp()` is used to ensure that the warp is converged and all writes are visible to other threads. The mask passed to `__syncwarp()` is computed using `__ballot_sync()`, which enables the bits for the first 16 threads in `mask`. Finally, the first thread (index 0) computes the sum over all initialized shared memory elements and writes it to global memory.

Building the application with `-G` to enable debug information and running it under the synccheck tool on SM 7.0 and above, multiple errors like the following are reported:

```
$ make run_illegal_syncwarp
========= Barrier error detected. Invalid arguments
=========      at __cuda_sm70_warpsync+0x30
=========      by thread (0,0,0) in block (0,0,0)
=========      Device Frame:__syncwarp(unsigned int)+0xf0 in /usr/local/cuda/
include/sm_30_intrinsics.hpp:110
=========      Device Frame:myKernel(int *)+0x3c0 in /home/cuda/github/compute-
sanitizer-samples/Synccheck/illegal_syncwarp.cu:48
```

The issue is with the `__syncwarp(mask)` at line 48. All threads for which `tx < (NumThreads / 2)` holds true are enabled in the mask, which are threads 0-15. However, the if condition evaluates true for threads 0-16. As a result, thread 16 executes the `__syncwarp(mask)` but does not declare itself in the mask parameter as required.

11.5. Example Use of suppressions

This section presents two example usages of the suppressions feature of Compute Sanitizer. The first example displays an API suppression (in the memcheck tool). The second example displays an initcheck report suppression.

11.5.1. API error suppression

The API error application can be found on the compute-sanitizer github repository

This application can be compiled using the provided Makefile:

```
make
```

In this example, we have a simple loop where the application attempts to allocate a large decreasing size. We can expect the cudaMalloc API to fail several times before the size is small enough to fit on the GPU.

```
$ make run_memcheck
 /usr/local/cuda/compute-sanitizer/compute-sanitizer suppressions_demo
 ========= COMPUTE-SANITIZER
 ========= Program hit cudaErrorMemoryAllocation (error 2) due to "out of
 memory" on CUDA API call to cudaMalloc.
```

In order to generate a suppressions file, we need to use the **--xml** option combined with the **--save** option for the output file name. Running that command still prints out error as before, but it also creates an XML file and populates it with a record of the output.

```
$ make gen_supp
 /usr/local/cuda/compute-sanitizer/compute-sanitizer --save supp.xml --xml
 suppressions_demo
 ========= COMPUTE-SANITIZER
 ========= Program hit cudaErrorMemoryAllocation (error 2) due to "out of
 memory" on CUDA API call to cudaMalloc.
 [...]
$ cat supp.xml

<?xml version="1.0" encoding="utf-8"?>
  <ComputeSanitizerOutput>
    <record>
      <kind>Api</kind>
      <what>
        <text>Program hit cudaErrorMemoryAllocation (error 2) due to out of
memory on CUDA API call to cudaMalloc.</text>
      <api>cudaMalloc</api>
      <error>cudaErrorMemoryAllocation</error>
      <message>out of memory</message>
      <result>2</result>
    </what>
    <hostStack>
    [...]
    </hostStack>
```

Now, we can use that file as input to run the tool, along with the **--suppressions** option to ignore that error.

```
$ make run_memcheck_with_supp
/usr/local/cuda/compute-sanitizer/compute-sanitizer --suppressions supp.xml
 suppressions_demo
========= COMPUTE-SANITIZER
========= ERROR SUMMARY: 0 errors
```

The XML file can be edited to change which errors are ignored. For instance, a regular expression can be used in the api tag to suppress a range of API calls. For instance `cuda.*` will ignore any errors related to an API starting with cuda.

Other tags that can be edited are the result and hoststack ones. Note that the host stack appears in reverse order and the suppressions feature will compare every stack frame that was recorded.

11.5.2. Initcheck error suppression

The API error application can be found on the compute-sanitizer github repository

This application can be compiled using the provided Makefile:

```
make
```

In this example, we have a simple multiplication kernel. A call to `cudaMemset` is used to initialize the device memory to 0. However, it does not initialize the last byte of the array. The initcheck tool detects the unitialized access:

```
$ make run_initcheck
/usr/local/cuda/compute-sanitizer/compute-sanitizer --tool initcheck
 suppressions_initcheck_demo
========= COMPUTE-SANITIZER
========= Uninitialized __global__ memory read of size 4 bytes
=========     at mult(int *, int *, int)+0x60
=========     by thread (122,0,0) in block (0,0,0)
=========     Address 0x7f936fa001e8
[...]
========= ERROR SUMMARY: 1 error
```

In a similar fashion to the previous example, we can use the `--xml` option to generate a suppression file.

```
$ make initcheck_gen_supp
/usr/local/cuda/compute-sanitizer/compute-sanitizer --tool initcheck --save
 supp.xml --xml suppressions_initcheck_demo
========= COMPUTE-SANITIZER
========= Uninitialized __global__ memory read of size 4 bytes
[...]
```

Now, the error can be ignored using the XML file as input to the suppressions feature.

```
$ make run_initcheck_with_supp
/usr/local/cuda/compute-sanitizer/compute-sanitizer --tool initcheck --
suppressions supp.xml suppressions_initcheck_demo
========= COMPUTE-SANITIZER
========= ERROR SUMMARY: 0 errors
```

As with the API suppressions, the XML file can be edited to make the suppressions detection more generic, by editing or removing the threadId, blockId, size and device stack tags.