CUDA-MEMCHECK

User Manual

TABLE OF CONTENTS

LIST OF TABLES

Chapter 1.
INTRODUCTION

1.1 About CUDA-MEMCHECK

CUDA-MEMCHECK is a functional correctness checking suite included in the CUDA toolkit. This suite contains multiple tools that can perform different types 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 also reports hardware exceptions encountered by the GPU. The *racecheck* tool can report shared memory data access hazards that can cause data races.This document describes the usage of these tools.

CUDA-MEMCHECK can be run in *standalone mode* where the user's application is started under CUDA-MEMCHECK. The *memcheck* tool can also be enabled in *integrated mode* inside CUDA-GDB.

1.2 Why CUDA-MEMCHECK?

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 errors that are hard to detect and time consuming to debug. The number of such errors increases substantially when dealing with thousands of threads. The CUDA-MEMCHECK suite is designed to detect such errors in your CUDA application. Using the memcheck tool, CUDA-MEMCHECK can identify memory access errors as well as hardware reported program errors. The racecheck tool in CUDA-MEMCHECK can identify hazards caused by race conditions in the CUDA program.

1.3 New Features in 5.0

▸ Reporting of data access hazards in shared memory accesses. This is supported on Fermi SM 2.x and Kepler SM 3.0 GPUs. This functionality is not supported on Windows XP. For more information, see Racecheck Tool.

▸ Support for SM 3.0 and SM 3.5 GPUs. For more information, see Supported Devices.

▸ Support for dynamic parallelism. All memory access error detection is supported for applications using dynamic parallelism. For more information, see Dynamic Parallelism.

▸ Precise error detection for local loads/stores, shared loads/stores, global atomics/reductions. On SM 3.5, added precise memory access error detection for noncoherent global loads through the texture unit. For more information, see Memory Access Error Reporting.

▸ Error detection in device side malloc()/free(), such as double free() or invalid free() on the GPU. For more information, see Device Side Allocation Checking.

▸ Leak checking for allocations on the device heap. For more information, see Leak Checking.

▸ Display of a saved stack backtrace on the host and captured backtrace on the device for different errors. For more information, see Stack Backtraces.

▸ Reporting of CUDA API errors in the user's application. For more information, see CUDA API Error Checking.

▸ Added display of mangled, demangled, and full prototype of the kernel. For more information, see Name Demangling.

▸ Increased functionality in integrated mode with CUDA-GDB. Added reporting of the address and address space being accessed that caused a precise exception. Added checking of device side malloc() and free() when in integrated mode. For more information, see Integrated Mode.

▸ Support for applications compiled separately that use the device side linker.

▸ Support for applications compiled with the -lineinfo flag.

▸ New style of command line options. For more information, see Command Line Options.

▸ Changed default behavior. CUDA-MEMCHECK will display backtraces by default and will report API errors by default. For more information, see Command Line Options.

1.4 How to Get CUDA-MEMCHECK

CUDA-MEMCHECK is installed as part of the CUDA toolkit.

1.5 CUDA-MEMCHECK tools

Tools allow use the basic CUDA-MEMCHECK infrastructure to provide different checking mechanisms. 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

Table 1 Supported Modes by CUDA-MEMCHECK tool

Tool Name	Standalone Mode	Integrated Mode
Memcheck	Yes	Yes
Racecheck	Yes	No

Chapter 2.
USING CUDA-MEMCHECK

CUDA-MEMCHECK tools can be invoked by running the `cuda-memcheck` executable as follows:

```
cuda-memcheck [options] app_name [app_options]
```

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

2.1 Command Line Options

Command line options can be specified to `cuda-memcheck`. With some exceptions, the options to memcheck are usually of the form `--option value`. The option list can be terminated by specifying `--`. All subsequent words on the command line 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 as passed to CUDA-MEMCHECK. Some options have a one character short form, which is given in parentheses. These options can be invoked using a single hypen. 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, this is shown below in braces {}. An option that can accept any numerical value is represented as *{number}* . Blank entries indicate that the value is not present.

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 2 CUDA-MEMCHECK Command line options

Option	Values	Default	Description
demangle	full, simple, no	full	Enables demangling of device function names. For more information, see Name Demangling.

Option	Values	Default	Description
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 memcheck will return if the original application succeeded but there were any memcheck detected errors. This is meant to allow CUDA-MEMCHECK to be integrated into automated test suites
flush-to-disk	yes,no	no	Forces every disk write to be flushed to disk. When enabled, this will make CUDA-MEMCHECK tools much slower.
force-blocking-launches	yes,no	no on SM 2.0+, yes on SM 1.x	This forces all host kernel launches to be sequential. When enabled, the number and precision of memcheck reported errors will decrease. This option only has effect on SM 2.0 and higher GPUs
help (h)	N/A	N/A	Displays the help message
prefix	{string}	========	The string prepended to CUDA-MEMCHECK output lines
print-level	info,warn,error,fatal	info	The minimum level print level of messages from CUDA-MEMCHECK.
read	{filename}	N/A	The input memcheck file to read data from. This can be used in conjunction with the save option to allow processing records after a run.
save	{filename}	N/A	Filename where CUDA-MEMCHECK will save the output from the current run.
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.
tool	memcheck, racecheck	memcheck	Controls which CUDA-MEMCHECK tool is actively running
version (V)	N/A	N/A	Prints the version of cuda-memcheck

Table 3 *Memcheck* Tool Command line options

Option	Values	Default	Description
check-device-heap	yes,no	yes on SM 2.0+, no on SM 1.x	Enable checking of device heap allocations. This applies to both error checking and leak checking. This option only has effect on SM 2.0 and higher GPUs

Option	Values	Default	Description
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.
report-api-errors	yes,no	yes	Report errors if any CUDA API call fails. For more information, see CUDA API Error Checking.

2.2 Supported Operating Systems

The standalone CUDA-MEMCHECK binary is supported on all CUDA supported platforms i.e. Windows, Mac OS X, and supported Linux distributions. CUDA-MEMCHECK can interoperate with CUDA-GDB on Linux and Mac OS X.

2.3 Supported Devices

The CUDA-MEMCHECK tool suite is supported on all CUDA capable GPUs with SM versions 1.1 and above. Individual tools may support a different subset of GPUs. The table below contains the list of current tools and their supported GPUs

Table 4 Supported Devices by CUDA-MEMCHECK tool

Tool Name	SM 1.x	SM 2.x	SM 3.0	SM 3.5
Memcheck	Yes	Yes	Yes	Yes
Racecheck	No	Yes	Yes	No

2.4 Compilation Options

The CUDA-MEMCHECK tools *memcheck* and *racecheck* do not need any special compilation flags to function.

The output displayed by the CUDA-MEMCHECK 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 option to nvcc can be used. The CUDA-MEMCHECK 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 CUDA-MEMCHECK 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. 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
```

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, shared and global atomic instructions 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 5 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	Memory Access Error Reporting
Hardware exception	Errors that are reported by the hardware error reporting mechanism.	Device	Imprecise	Hardware Exception Reporting
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

Name	Description	Location	Precision	See also
cudaMalloc memory leaks	Allocations of device memory using `cudaMalloc()` that have not been freed by the application.	Host	Precise	Leak Checking
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 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. Imprecise error reporting is only supported on SM 2.0 and higher GPUs.

3.3 Using Memcheck

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

```
cuda-memcheck [memcheck_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 explictly 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
=========     at 0x00000060 in
 memcheck_demo.cu:6:unaligned_kernel(void)
=========     by thread (0,0,0) in block (0,0,0)
=========     Address 0x400100001 is misaligned
```

Let us examine this error line by line :

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

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 0x00000060 in memcheck_demo.cu:6:unaligned_kernel(void)
```

The second line contains the PC of the instruction, the source file and line number (if available) and the CUDA kernel name. In this example, the instruction causing the access was at PC 0x60 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 0x400100001 is misaligned
```

The fourth line contains the memory address being accessed and the type of 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 0x400100001 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 0x000079d8
=========       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 0x000079d8
```

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 follows :

```
========= 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 error 11 on CUDA API call to cudaMemset
```

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 Integrated Mode

You can execute the memcheck tool from within CUDA-GDB by using the following option before running the application:

```
(cuda-gdb) set cuda memcheck on
```

In integrated mode, the memcheck tool improves the precision of error reporting by CUDA-GDB. The memory access checks are enabled, allowing identification of the thread that may be causing a warp or device level exception.

3.6 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, CUDA-MEMCHECK 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.

3.7 Device Side Allocation Checking

On SM 2.0 and higher GPUs, 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 current implementation does not require space on the device heap, and so the heap allocation behavior of the program with and without memcheck should remain similar. The *memcheck* tool does require space in device global memory to track these heap allocations and will print an internal error message if it is not able to allocate this space in device global memory.

In addition to access checks, the *memcheck* tool can now perform libc style checks on the `malloc()`/`free()` calls. The tool will report an error if the application calls a `free()` twice on a kernel, 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

3.8 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. Starting in CUDA 5, allocations that are created dynamically on the device heap by calling `malloc()` inside a kernel are also tracked.

For an accurate leak checking summary to be generated, the application's CUDA context must be destroyed at the end. This can be done explicitly by calling `cuCtxDestroy()` in applications using the CUDA driver API, or by calling `cudaDeviceReset()` in applications programmed against the CUDA run time API.

The `--leak-check full` option must be specified to enable leak checking.

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 in 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. For supported architectures, see Supported Devices.

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 nondeterministic 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.

▶ Read-After-Write (*RAW*) 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.

▶ Write-After-Read (*WAR*) 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 CUDA-MEMCHECK application with the `--tool racecheck` option.

```
cuda-memcheck --tool racecheck [memcheck_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.

> 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 Understanding Racecheck Reports

The racecheck tool produces a series of messages detailing information about hazards in the application. The tool is byte accurate and so 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 0x00000088 in
 raceWAW.cu:18:WAW(void)
=========        Write Thread (1, 0, 0) at 0x00000088 in
 raceWAW.cu:18:WAW(void)
=========        Current Value : 0, 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 its 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. This can be

▸ *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.
▸ ERROR : The highest level of severity. Correspond 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.

In this case, the message is at the ERROR level of severity.

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 0x00000088 in raceWAW.cu:18:WAW(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 the thread block. In this case, the thread is at index (0,0,0). Following this, we have the byte offset of the instruction which did the access in the kernel. In this example, the offset is 0x88. This is followed by the source file and line number (if line number information is available). The final item on this line is the name of the kernel that was being executed.

The third line contains similar information about the second thread which 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 : 0, 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 0 to the shared memory location. The second thread is attempting to write the value 2.

Chapter 5.
CUDA-MEMCHECK FEATURES

5.1 Nonblocking Mode

By default, on SM 2.0 and higher GPUs the standalone CUDA-MEMCHECK 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. Blocking launch mode is always enabled on Mac OS X 10.6 and on Windows XP. This flag has no effect on GPUs less than SM 2.0. One side effect is that when in blocking mode, only the first thread to hit an error in a kernel will be reported.

5.2 Stack Backtraces

In standalone mode, CUDA-MEMCHECK can generate backtraces when given `--show-backtrace` option. Backtraces usually consist of two sections - a saved host backtrace that leads upto the CUDA driver call site, and a device backtrace at the time of the error. Each backtrace contains a list of function calls 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

In CUDA 5, the host stack backtrace will show a maximum of 61 frames.

Backtraces are printed for most CUDA-MEMCHECK 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 6 CUDA-MEMCHECK Stack Backtrace Information

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

5.3 Name Demangling

The CUDA-MEMCHECK suite now 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.

5.4 Dynamic Parallelism

The CUDA-MEMCHECK 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 and as a result, in memory constrained environments, *memcheck* may fail to initialize with an internal out of memory error.

For limitations, see Known Issues.

5.5 Error Actions

On encountering an error, CUDA-MEMCHECK behavior depends on the type of error. The default behavior of CUDA-MEMCHECK is to continue execution on purely

---

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

host side errors. Hardware exceptions detected by the memcheck tool cause the CUDA context to 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.

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 7 CUDA-MEMCHECK Error Actions

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

Chapter 6.
CUDA-MEMCHECK TOOL EXAMPLES

6.1 Example Use of Memcheck

This section presents a walk-through of running the memcheck tool from CUDA-MEMCHECK on a simple application called `memcheck_demo`.

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

memcheck_demo.cu source code

```
#include <stdio.h>

__device__ int x;

__global__ void unaligned_kernel(void) {
    *(int*) ((char*)&x + 1) = 42;
}

__device__ void out_of_bounds_function(void) {
    *(int*) 0x87654320 = 42;
}

__global__ void out_of_bounds_kernel(void) {
    out_of_bounds_function();
}

void run_unaligned(void) {
    printf("Running unaligned_kernel\n");
    unaligned_kernel<<<1,1>>>();
    printf("Ran unaligned_kernel: %s\n",
    cudaGetErrorString(cudaGetLastError()));
    printf("Sync: %s\n", cudaGetErrorString(cudaThreadSynchronize()));
}

void run_out_of_bounds(void) {
    printf("Running out_of_bounds_kernel\n");
    out_of_bounds_kernel<<<1,1>>>();
    printf("Ran out_of_bounds_kernel: %s\n",
    cudaGetErrorString(cudaGetLastError()));
    printf("Sync: %s\n", cudaGetErrorString(cudaThreadSynchronize()));
}
```

```
int main() {
    int *devMem;

    printf("Mallocing memory\n");
    cudaMalloc((void**)&devMem, 1024);

    run_unaligned();
    run_out_of_bounds();

    cudaDeviceReset();
    cudaFree(devMem);

    return 0;
}
```

This application is compiled for release builds as :

```
nvcc -gencode arch=compute_20,code=sm_20 -o memcheck_demo
 memcheck_demo.cu
```

6.1.1 `memcheck_demo` Output

When a CUDA application causes access violations, the kernel launch may terminate with an error code of unspecified launch failure or a subsequent cudaThreadSynchronize call which will fail with an error code of unspecified launch failure.

This sample application is causing two failures but there is no way to detect where these kernels are causing the access violations, as illustrated in the following output:

```
$ ./memcheck_demo
Mallocing memory
Running unaligned_kernel
Ran unaligned_kernel: no error
Sync: unspecified launch failure
Running out_of_bounds_kernel
Ran out_of_bounds_kernel: unspecified launch failure
Sync: unspecified launch failure
```

6.1.2 `memcheck_demo` Output with Memcheck (Release Build)

In this case, since the application is built in release mode, the CUDA-MEMCHECK 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. In addition, CUDA API errors are reported, such as the invalid cudaFree() call in the application.

```
$ cuda-memcheck ./memcheck_demo
========= CUDA-MEMCHECK
Mallocing memory
```

```
Running unaligned_kernel
Ran unaligned_kernel: no error
Sync: no error
Running out_of_bounds_kernel
Ran out_of_bounds_kernel: no error
Sync: no error
========= Invalid __global__ write of size 4
=========     at 0x00000028 in unaligned_kernel(void)
=========     by thread (0,0,0) in block (0,0,0)
=========     Address 0x400100001 is misaligned
=========     Saved host backtrace up to driver entry point at
 kernel launch time
=========     Host Frame:/usr/local/lib/libcuda.so
 (cuLaunchKernel + 0x3ae) [0xddbee]
=========     Host Frame:/usr/local/lib/libcudart.so.5.0 [0xcd27]
=========     Host Frame:/usr/local/lib/libcudart.so.5.0
 (cudaLaunch + 0x1bb) [0x3778b]
=========     Host Frame:memcheck_demo [0xdfc]
=========     Host Frame:memcheck_demo [0xc76]
=========     Host Frame:memcheck_demo [0xc81]
=========     Host Frame:memcheck_demo [0xb03]
=========     Host Frame:memcheck_demo [0xc27]
=========     Host Frame:/lib64/libc.so.6 (__libc_start_main +
 0xfd) [0x1eb1d]
=========     Host Frame:memcheck_demo [0x9b9]
=========
========= Invalid __global__ write of size 4
=========     at 0x00000010 in out_of_bounds_kernel(void)
=========     by thread (0,0,0) in block (0,0,0)
=========     Address 0xffffffff87654320 is out of bounds
=========     Saved host backtrace up to driver entry point at
 kernel launch time
=========     Host Frame:/usr/local/lib/libcuda.so
 (cuLaunchKernel + 0x3ae) [0xddbee]
=========     Host Frame:/usr/local/lib/libcudart.so.5.0 [0xcd27]
=========     Host Frame:/usr/local/lib/libcudart.so.5.0
 (cudaLaunch + 0x1bb) [0x3778b]
=========     Host Frame:memcheck_demo [0xdfc]
=========     Host Frame:memcheck_demo [0xca0]
=========     Host Frame:memcheck_demo [0xcab]
=========     Host Frame:memcheck_demo [0xbbc]
=========     Host Frame:memcheck_demo [0xc2c]
=========     Host Frame:/lib64/libc.so.6 (__libc_start_main +
 0xfd) [0x1eb1d]
=========     Host Frame:memcheck_demo [0x9b9]
=========
========= Program hit error 17 on CUDA API call to cudaFree
=========     Saved host backtrace up to driver entry point at
 error
=========     Host Frame:/usr/local/lib/libcuda.so [0x28f850]
=========     Host Frame:/usr/local/lib/libcudart.so.5.0
 (cudaFree + 0x20d) [0x364ed]
=========     Host Frame:memcheck_demo [0xc3d]
=========     Host Frame:/lib64/libc.so.6 (__libc_start_main +
 0xfd) [0x1eb1d]
```

```
=========        Host Frame:memcheck_demo [0x9b9]
=========
========= ERROR SUMMARY: 3 errors
```

6.1.3 `memcheck_demo` Output with Memcheck (Debug Build)

The application is now built with device side debug information and function symbols as :

```
nvcc -G -Xcompiler -rdynamic -gencode arch=compute_20,code=sm_20
 -o memcheck_demo memcheck_demo.cu
```

Now run this application with CUDA-MEMCHECK 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 CUDA-MEMCHECK detected the error. The application continued to run the second kernel. The error detected in the second kernel causes it to terminate early. Finally, the application calls cudaDeviceReset(), which destroys the CUDA context and then attempts to call cudaFree(). This call returns an API error that is caught and displayed by memcheck.

```
$ cuda-memcheck ./memcheck_demo
========= CUDA-MEMCHECK
Mallocing memory
Running unaligned_kernel
Ran unaligned_kernel: no error
Sync: no error
Running out_of_bounds_kernel
Ran out_of_bounds_kernel: no error
Sync: no error
========= Invalid __global__ write of size 4
=========     at 0x00000028 in
 memcheck_demo.cu:6:unaligned_kernel(void)
=========     by thread (0,0,0) in block (0,0,0)
=========     Address 0x400100001 is misaligned
=========     Saved host backtrace up to driver entry point at
 kernel launch time
=========     Host Frame:/usr/local/lib/libcuda.so
 (cuLaunchKernel + 0x3ae) [0xddbee]
=========     Host Frame:/usr/local/lib/libcudart.so.5.0 [0xcd27]
=========     Host Frame:/usr/local/lib/libcudart.so.5.0
 (cudaLaunch + 0x1bb) [0x3778b]
=========     Host Frame:memcheck_demo
 (_Z10cudaLaunchIcE9cudaErrorPT_ + 0x18) [0x11a4]
=========     Host Frame:memcheck_demo
 (_Z35__device_stub__Z16unaligned_kernelvv + 0x1d) [0x101d]
=========     Host Frame:memcheck_demo (_Z16unaligned_kernelv +
 0x9) [0x1028]
```

```
=========        Host Frame:memcheck_demo (_Z13run_unalignedv +
 0x76) [0xeaa]
=========        Host Frame:memcheck_demo (main + 0x28) [0xfce]
=========        Host Frame:/lib64/libc.so.6 (__libc_start_main +
 0xfd) [0x1eb1d]
=========        Host Frame:memcheck_demo [0xd79]
=========
========= Invalid __global__ write of size 4
=========        at 0x00000028 in
 memcheck_demo.cu:10:out_of_bounds_function(void)
=========        by thread (0,0,0) in block (0,0,0)
=========        Address 0x87654320 is out of bounds
=========        Device
 Frame:memcheck_demo.cu:15:out_of_bounds_kernel(void)
 (out_of_bounds_kernel(void) : 0x10)
=========        Saved host backtrace up to driver entry point at
 kernel launch time
=========        Host Frame:/usr/local/lib/libcuda.so
 (cuLaunchKernel + 0x3ae) [0xddbee]
=========        Host Frame:/usr/local/lib/libcudart.so.5.0 [0xcd27]
=========        Host Frame:/usr/local/lib/libcudart.so.5.0
 (cudaLaunch + 0x1bb) [0x3778b]
=========        Host Frame:memcheck_demo
 (_Z10cudaLaunchIcE9cudaErrorPT_ + 0x18) [0x11a4]
=========        Host Frame:memcheck_demo
 (_Z39__device_stub__Z20out_of_bounds_kernelvv + 0x1d) [0x1047]
=========        Host Frame:memcheck_demo (_Z20out_of_bounds_kernelv
 + 0x9) [0x1052]
=========        Host Frame:memcheck_demo (_Z17run_out_of_boundsv +
 0x76) [0xf63]
=========        Host Frame:memcheck_demo (main + 0x2d) [0xfd3]
=========        Host Frame:/lib64/libc.so.6 (__libc_start_main +
 0xfd) [0x1eb1d]
=========        Host Frame:memcheck_demo [0xd79]
=========
========= Program hit error 17 on CUDA API call to cudaFree
=========        Saved host backtrace up to driver entry point at
 error
=========        Host Frame:/usr/local/lib/libcuda.so [0x28f850]
=========        Host Frame:/usr/local/lib/libcudart.so.5.0
 (cudaFree + 0x20d) [0x364ed]
=========        Host Frame:memcheck_demo (main + 0x3e) [0xfe4]
=========        Host Frame:/lib64/libc.so.6 (__libc_start_main +
 0xfd) [0x1eb1d]
=========        Host Frame:memcheck_demo [0xd79]
=========
========= ERROR SUMMARY: 3 errors
```

6.1.4 Leak Checking in CUDA-MEMCHECK

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 CUDA-
MEMCHECK.

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 CUDA-MEMCHECK still prints errors it encountered while running the application.

```
$ cuda-memcheck --leak-check full memcheck_demo
========= CUDA-MEMCHECK
Mallocing memory
Running unaligned_kernel
Ran unaligned_kernel: no error
Sync: no error
Running out_of_bounds_kernel
Ran out_of_bounds_kernel: no error
Sync: no error
========= Invalid __global__ write of size 4
=========     at 0x00000060 in
 memcheck_demo.cu:6:unaligned_kernel(void)
=========     by thread (0,0,0) in block (0,0,0)
=========     Address 0x400100001 is misaligned
=========     Saved host backtrace up to driver entry point at
 kernel launch time
=========     Host Frame:/usr/local/lib/libcuda.so
 (cuLaunchKernel + 0x3ae) [0xddbee]
=========     Host Frame:/usr/local/lib/libcudart.so.5.0 [0xcd27]
=========     Host Frame:/usr/local/lib/libcudart.so.5.0
 (cudaLaunch + 0x1bb) [0x3778b]
=========     Host Frame:memcheck_demo
 (_Z10cudaLaunchIcE9cudaErrorPT_ + 0x18) [0x122c]
=========     Host Frame:memcheck_demo
 (_Z35__device_stub__Z16unaligned_kernelvv + 0x1d) [0x10a6]
=========     Host Frame:memcheck_demo (_Z16unaligned_kernelv +
 0x9) [0x10b1]
=========     Host Frame:memcheck_demo (_Z13run_unalignedv +
 0x76) [0xf33]
=========     Host Frame:memcheck_demo (main + 0x28) [0x1057]
=========     Host Frame:/lib64/libc.so.6 (__libc_start_main +
 0xfd) [0x1eb1d]
=========     Host Frame:memcheck_demo [0xde9]
=========
========= Invalid __global__ write of size 4
=========     at 0x00000028 in
 memcheck_demo.cu:10:out_of_bounds_function(void)
=========     by thread (0,0,0) in block (0,0,0)
=========     Address 0x87654320 is out of bounds
=========     Device
 Frame:memcheck_demo.cu:15:out_of_bounds_kernel(void)
 (out_of_bounds_kernel(void) : 0x10)
```

```
=========        Saved host backtrace up to driver entry point at
 kernel launch time
=========        Host Frame:/usr/local/lib/libcuda.so
 (cuLaunchKernel + 0x3ae) [0xddbee]
=========        Host Frame:/usr/local/lib/libcudart.so.5.0 [0xcd27]
=========        Host Frame:/usr/local/lib/libcudart.so.5.0
 (cudaLaunch + 0x1bb) [0x3778b]
=========        Host Frame:memcheck_demo
 (_Z10cudaLaunchIcE9cudaErrorPT_ + 0x18) [0x122c]
=========        Host Frame:memcheck_demo
 (_Z39__device_stub__Z20out_of_bounds_kernelvv + 0x1d) [0x10d0]
=========        Host Frame:memcheck_demo (_Z20out_of_bounds_kernelv
 + 0x9) [0x10db]
=========        Host Frame:memcheck_demo (_Z17run_out_of_boundsv +
 0x76) [0xfec]
=========        Host Frame:memcheck_demo (main + 0x2d) [0x105c]
=========        Host Frame:/lib64/libc.so.6 (__libc_start_main +
 0xfd) [0x1eb1d]
=========        Host Frame:memcheck_demo [0xde9]
=========
========= Leaked 1024 bytes at 0x400200000
=========        Saved host backtrace up to driver entry point at
 cudaMalloc time
=========        Host Frame:/usr/local/lib/libcuda.so (cuMemAlloc_v2
 + 0x236) [0xe9746]
=========        Host Frame:/usr/local/lib/libcudart.so.5.0
 [0x26dd7]
=========        Host Frame:/usr/local/lib/libcudart.so.5.0 [0xb37b]
=========        Host Frame:/usr/local/lib/libcudart.so.5.0
 (cudaMalloc + 0x17a) [0x36e6a]
=========        Host Frame:memcheck_demo (main + 0x23) [0x1052]
=========        Host Frame:/lib64/libc.so.6 (__libc_start_main +
 0xfd) [0x1eb1d]
=========        Host Frame:memcheck_demo [0xde9]
=========
========= Program hit error 17 on CUDA API call to cudaFree
=========        Saved host backtrace up to driver entry point at
 error
=========        Host Frame:/usr/local/lib/libcuda.so [0x28f850]
=========        Host Frame:/usr/local/lib/libcudart.so.5.0
 (cudaFree + 0x20d) [0x364ed]
=========        Host Frame:memcheck_demo (main + 0x3e) [0x106d]
=========        Host Frame:/lib64/libc.so.6 (__libc_start_main +
 0xfd) [0x1eb1d]
=========        Host Frame:memcheck_demo [0xde9]
=========
========= LEAK SUMMARY: 1024 bytes leaked in 1 allocations
========= ERROR SUMMARY: 3 errors
```

6.2 Integrated CUDA-MEMCHECK Example

This example shows how to enable CUDA-MEMCHECK from within CUDA-GDB and how to detect errors within the debugger so you can access the line number information and check the state of the variables

In this example the unaligned kernel has a misaligned memory access in block 1 lane 1, which gets trapped as an illegal lane address at line 6 from within CUDA-GDB. Note that CUDA-GDB displays the address and that caused the bad access.

```
(cuda-gdb) set cuda memcheck on
(cuda-gdb) run
Starting program: memcheck_demo
[Thread debugging using libthread_db enabled]
Mallocing memory
[New Thread 0x7ffff6fe1710 (LWP 7783)]
[Context Create of context 0x6218a0 on Device 0]
[Launch of CUDA Kernel 0 (memset32_post<<<(1,1,1),(64,1,1)>>>) on
 Device 0]
Running unaligned_kernel
[Launch of CUDA Kernel 1 (unaligned_kernel<<<(1,1,1),(1,1,1)>>>)
 on Device 0]
Memcheck detected an illegal access to address
 (@global)0x400100001

Program received signal CUDA_EXCEPTION_1, Lane Illegal Address.
[Switching focus to CUDA kernel 1, grid 2, block (0,0,0), thread
 (0,0,0), device 0, sm 0, warp 0, lane 0]
0x000000000078b8b0 in unaligned_kernel<<<(1,1,1),(1,1,1)>>> () at
 memcheck_demo.cu:6
6          *(int*) ((char*)&x + 1) = 42;
(cuda-gdb) print &x
$1 = (@global int *) 0x400100000
(cuda-gdb) continue
Continuing.
[Termination of CUDA Kernel 1 (unaligned_kernel<<<(1,1,1),
(1,1,1)>>>) on Device 0]
[Termination of CUDA Kernel 0 (memset32_post<<<(1,1,1),
(64,1,1)>>>) on Device 0]

Program terminated with signal CUDA_EXCEPTION_1, Lane Illegal
 Address.
The program no longer exists.
(cuda-gdb)
```

Appendix A.
MEMORY ACCESS ERROR REPORTING

The memcheck tool will report memory access errors when run standalone or in integrated mode with CUDA-GDB. The table below describes the types of accesses that are checked and the SM version where such checks happen

Table 8 Memcheck memory accesss error detection support

Error Type	SM 1.x	SM 2.x	SM 3.0	SM 3.5
Global	Yes	Yes	Yes	Yes
Shared	Yes	Yes	Yes	Yes
Local	Yes	Yes	Yes	Yes
Global Atomic	Yes	Yes	Yes	Yes
Load through texture	N/A	N/A	N/A	Yes

Appendix B.
HARDWARE EXCEPTION REPORTING

The CUDA-MEMCHECK tool will report hardware exceptions when run as a standalone or as part of CUDA-GDB. The table below enumerates the supported exceptions, their precision and scope, as well as a brief description of their cause. For more detailed information, see the documentation for CUDA-GDB.

Table 9 CUDA Exception Codes

Exception code	Precision of the Error	Scope of the Error	Description
CUDA_EXCEPTION_1 : "Lane Illegal Address"	Precise	Per lane/thread error	This occurs when a thread accesses an illegal (out of bounds) global address.
CUDA_EXCEPTION_2 : "Lane User StackOverflow"	Precise	Per lane/thread error	This occurs when a thread exceeds its stack memory limit.
CUDA_EXCEPTION_3: "Device Hardware Stack Overflow"	Not precise	Global error on the GPU	This occurs when the application triggers a global hardware stack overflow. The main cause of this error is large amounts of divergence in the presence of function calls.
CUDA_EXCEPTION_4: "Warp Illegal Instruction"	Not precise	Warp error	This occurs when any thread within a warp has executed an illegal instruction.
CUDA_EXCEPTION_5: "Warp Out-of-range Address"	Not precise	Warp error	This occurs when any thread within a warp accesses an address that is outside the valid range of local or shared memory regions.
CUDA_EXCEPTION_6: "Warp Misaligned Address"	Not precise	Warp error	This occurs when any thread within a warp accesses an address in the local or shared memory segments that is not correctly aligned.
CUDA_EXCEPTION_7: "Warp Invalid Address Space"	Not precise	Warp error	This occurs when any thread within a warp executes an instruction

Exception code	Precision of the Error	Scope of the Error	Description
			that accesses a memory space not permitted for that instruction.
CUDA_EXCEPTION_8: "Warp Invalid PC"	Not precise	Warp error	This occurs when any thread within a warp advances its PC beyond the 40-bit address space.
CUDA_EXCEPTION_9: "Warp Hardware Stack Overflow"	Not precise	Warp error	This occurs when any thread in a warp triggers a hardware stack overflow. This should be a rare occurrence.
CUDA_EXCEPTION_10: "Device Illegal Address"	Not precise	Global error	This occurs when a thread accesses an illegal (out of bounds) global address.
CUDA_EXCEPTION_11: "Lane Misaligned Address"	Precise	Per lane/thread error	This occurs when a thread accesses a global address that is not correctly aligned.
CUDA_EXCEPTION_12: "Warp Assert"	Precise	Per warp	This occurs when any thread in the warp hits a device side assertion.
CUDA_EXCEPTION_13: "Lane Syscall Error"	Precise	Per lane	This occurs when a particular thread causes an syscall error, such as calling `free()` in a kernel on an already free'd pointer.
"Unknown Exception"	Not precise	Global error	The precise cause of the exception is unknown. Potentially, this may be due to Device Hardware Stack overflows or a kernel generating an exception very close to its termination.

Appendix C.
KNOWN ISSUES

The following are known issues with the current release.

▶ Applications run much slower under CUDA-MEMCHECK tools. This may cause some kernel launches to fail with a launch timeout error when running with CUDA-MEMCHECK enabled.
▶ On Windows XP, the standalone CUDA-MEMCHECK tools will always run in blocking launch mode.
▶ When running CUDA-MEMCHECK tools in integrated mode with CUDA-GDB, only the *memcheck* tool is enabled. Also, the following features are disabled:

   ▶ Nonblocking launches
   ▶ Leak checking
   ▶ API error checking

▶ CUDA-MEMCHECK tools do not support CUDA/Direct3D interop.
▶ The memcheck tool does not support CUDA API error checking for API calls made on the GPU using dynamic parallelism.
▶ The racecheck tool does not support dynamic parallelism or SM 3.5.

www.nvidia.com