CUDA-MEMCHECK

DU-05355-001_v041 | January 10, 2012

User Manual

TABLE OF CONTENTS

01 INTRODUCTION

The CUDA toolkit includes a memory-checking tool for detecting and debugging memory errors in CUDA applications. This document describes that tool, called CUDA-MEMCHECK.

About CUDA-MEMCHECK

Why CUDA-MEMCHECK

NVIDIA simplifies the debugging of CUDA programming errors with its powerful CUDA-GDB hardware debugger. However, every programmer invariably encounters memory related errors that are hard to detect and time consuming to debug. The number of memory related errors increases substantially when dealing with thousands of threads. The CUDA-MEMCHECK tool is designed to detect such memory access errors in your CUDA application.

New Features in 4.1

- The CUDA-MEMCHECK tool now supports checking accesses to OpenGL interoperation allocations.
- On sm_20 and higher GPUs, the CUDA-MEMCHECK tool can detect misaligned and out-of-bounds accesses to regions on the device heap, i.e. memory allocated using malloc() inside a kernel.

  For more information, see "Device Side Allocation Checking" on page 4.
- On sm_20 and higher GPUs, the CUDA-MEMCHECK tool supports error reporting in applications launching concurrent kernels.

  For more information, see "Nonblocking Mode" on page 5.
- The CUDA-MEMCHECK tool supports reporting of leaked allocations. These are allocations create with cudaMalloc that do not have a corresponding cudaFree at the time the context was destroyed.

  For more information, see "Leak Checking" on page 5.
- The CUDA-MEMCHECK tool can save error records to a file and subsequently read and display them.
- On sm_20 and higher GPUs, the CUDA-MEMCHECK tool will report errors from all the threads that hit the error, rather than just the first thread.
- The CUDA-MEMCHECK tool will report and display messages if it was unable to perform error checking of the user's application due to internal errors.

Installation and Cross-Platform Support

The standalone CUDA-MEMCHECK binary gets installed with CUDA-GDB as part of the CUDA toolkit installation, and is supported on all CUDA supported platforms.

CUDA Memory Architecture

CUDA uses a segmented memory architecture that allows applications to access data in global, local, shared, constant, and texture memory.

A new unified addressing mode has been introduced in Fermi GPUs that allows data in global, local, and shared memory to be accessed with a generic 40-bit address.

02 CUDA-MEMCHECK FEATURES

Supported Error Detection

The CUDA-MEMCHECK tool supports detection of out-of-bounds and misaligned global memory accesses.

For sm_20 and higher GPUs, CUDA-MEMCHECK also detects hardware exceptions. The supported exceptions are enumerated in Appendix A.

Device Side Allocation Checking

On sm_20 and higher GPUs, the CUDA-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 by the standalone CUDA-MEMCHECK tool and can be disabled by specifying the **`--nodevheap`** flag. This feature is only activated if there is at least one kernel in the application that calls malloc().

The current implementation modifies the heap allocation, thus the total space on the device heap available for the user program may be affected. Furthermore, leak checking as described in the Leak Checking section is not supported for allocations on the device heap.

Nonblocking Mode

By default, on sm_20 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 **`--blocking`** flag. Blocking mode is always enabled on Mac OS X 10.6 and on Windows XP. This flag has no effect on GPUs less than sm_20. One side effect is that when in blocking mode, only the first thread to hit an error in a kernel will be reported.

Continue Mode

By default, the CUDA-MEMCHECK tool will stop the program execution after the first error inside a kernel is encountered. This can be overridden with the **`--continue`** flag to place the CUDA-MEMCHECK tool in continue mode.

In continue mode, the tool will allow the user application to continue, launching other kernels. An important note of caution is that if the user application depends on each kernel completing successfully, subsequent launches may produce incorrect data. On sm_20 and higher, GPUs receiving an imprecise hardware exception (as listed in Appendix A) will terminate the application, regardless of whether continue mode is set.

Continue mode is always implicitly enabled on MacOS X 10.6.

Leak Checking

In addition to detection out of bounds and misaligned accesses, the CUDA-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 CUDA-MEMCHECK tool tracks only device memory allocations created using the CUDA driver or runtime APIs. Allocations that are created dynamically on the device heap by calling malloc() inside a kernel are not included in the list.

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 programs using the CUDA driver API, or by calling cudaDeviceReset() in applications programmed against the CUDA run time API.

The **`--leakcheck`** option must be specified explicitly for leak checking to be enabled.

03 USING CUDA-MEMCHECK

You can run CUDA-MEMCHECK as either a standalone tool or as part of CUDA-GDB.

Differences Between Standalone and Integrated Mode

When running in integrated mode with CUDA-GDB, the CUDA-MEMCHECK tool has the following behavior:

- Error detection on the device heap is disabled
- Launches will happen in blocking mode
- Continue mode is disabled
- Leak checking is disabled

The following sections describe each approach to using CUDA-MEMCHECK:

- "Using Standalone CUDA-MEMCHECK" on page 7
- "Using Integrated CUDA-MEMCHECK" on page 13

Using Standalone CUDA-MEMCHECK

To run CUDA-MEMCHECK as a standalone tool, pass the application name as a parameter.

- Syntax:

  ```
  cuda-memcheck [options] [your-program] [your-program-options]
  ```

- Options field:
  - **-b | --blocking** Use blocking launches.
  - **-c | --continue** Try to continue running on memory access violations.
  - **-h | --help** Show this message.
  - **-l | --leakcheck** Print leak information for static allocations.
  - **-n | --nodevheap** Disable checking allocations on the device heap
  - **-p | --prefix string** Changes the prefix string displayed by CUDA-MEMCHECK.
  - **-r | --read file** Reads error records from a given file.
  - **-s | --save file** Saves the error record to file.
  - **-V| --version** Print the version of CUDA-MEMCHECK.

  Refer to "Known Issues" on page 17 regarding use of the **`-continue`** flag.

You can execute either a debug or release build of your CUDA application with CUDA-MEMCHECK.

- Using a debug version of your application built with the **`-g -G`** option pair gives you additional information regarding the line number of the access violation.
- With a release version of the application, CUDA-MEMCHECK logs only the name of the kernel responsible for the access violation.

Sample Application Outputs

This section presents a walk-through of a CUDA-MEMCHECK run with a simple application called **memcheck_demo**.

Note: 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;
}

__global__ void out_of_bounds_kernel(void) {
*(int*) 0x87654320 = 42;
}

int main() {
    int *devMem;

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

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

    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()));

    cudaDeviceReset();
    return 0;
}
```

Application output without CUDA-MEMCHECK

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
```

Application output with CUDA-MEMCHECK (Debug Build)

Now run this application with CUDA-MEMCHECK and check the output. We will use the **`--continue`** option to let CUDA-MEMCHECK continue executing the rest of the kernel after its first access violation.

In the output below the first kernel does not see the unspecified launch failure error since that was the only access violation that kernel executes, and with the **`--continue`** flag set, CUDA-MEMCHECK will force it to continue. Depending on the application error checking, with the **`--continue`** flag set CUDA-MEMCHECK can detect more than one occurrence of the errors across kernels, but reports only the first error per kernel.

```
$ cuda-memcheck --continue ./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 0x00000038 in memcheck_demo.cu:5:unaligned_kernel
=========     by thread (0,0,0) in block (0,0,0)
=========     Address 0x200200001 is misaligned
=========
========= Invalid __global__ write of size 4
=========     at 0x00000030 in memcheck_demo.cu:10:out_of_bounds_kernel
=========     by thread (0,0,0) in block (0,0,0)
=========     Address 0x87654320 is out of bounds
=========
```

```
=========
========= ERROR SUMMARY: 2 errors
```

Application output with CUDA-MEMCHECK, without --continue (Debug Build)

Now run this application with CUDA-MEMCHECK but without using the **--continue** option.

Without the **--continue** option, the first kernel shows the unspecified launch failure and only the first error gets reported by CUDA-MEMCHECK. In this case, after the access violation in the first kernel the application allows the second kernel to execute and there is application output for both kernels. Even so, the CUDA-MEMCHECK error is logged only for the first kernel.

```
$ cuda-memcheck  ./memcheck_demo
========= CUDA-MEMCHECK
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
========= Invalid __global__ write of size 4
=========     at 0x00000038 in memcheck_demo.cu:5:unaligned_kernel
=========     by thread (0,0,0) in block (0,0,0)
=========     Address 0x200200001 is misaligned
=========
=========
========= ERROR SUMMARY: 1 error
```

Application output with CUDA-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.

```
$ cuda-memcheck  ./memcheck_demo
========= CUDA-MEMCHECK
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
========= Invalid __global__ write of size 4
=========     at 0x00000028 in unaligned_kernel
=========     by thread (0,0,0) in block (0,0,0)
=========     Address 0x200200001 is misaligned
=========
=========
========= ERROR SUMMARY: 1 error
```

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 **`--leakcheck`** flag to CUDA-MEMCHECK.

When running the program with the leak check flag, 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. 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 --leakcheck ./memcheck_demo
========= CUDA-MEMCHECK
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
========= Invalid __global__ write of size 4
=========     at 0x00000028 in unaligned_kernel
=========     by thread (0,0,0) in block (0,0,0)
=========     Address 0x200200001 is misaligned
=========
========= Leaked 1024 bytes at 0x200300000
=========
========= LEAK SUMMARY: 1024 bytes leaked in 1 allocations
========= ERROR SUMMARY: 1 error
```

Using Integrated CUDA-MEMCHECK

You can execute CUDA-MEMCHECK from within CUDA-GDB by using the following variable before running the application:

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

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 5 from within CUDA-GDB.

```
(cuda-gdb) run
Starting program: memcheck_demo
[Thread debugging using libthread_db enabled]
[New process 23653]
Running unaligned_kernel
[New Thread 140415864006416 (LWP 23653)]
[Launch of CUDA Kernel 0 on Device 0]

Program received signal CUDA_EXCEPTION_1, Lane Illegal Address.
[Switching to CUDA Kernel 0 (<<<(0,0),(0,0,0)>>>)]
0x0000000000992e68 in unaligned_kernel <<<(1,1),(1,1,1)>>> () at
memcheck_demo.cu:5
5               *(int*) ((char*)&x + 1) = 42;
(cuda-gdb) print &x
$1 = (@global int *) 0x42c00
(cuda-gdb) continue
Continuing.

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

APPENDIX A 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 A.1 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 Stack Overflow"	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.

Table A.1 CUDA Exception Codes (continued)

Exception code	Precision of the Error	Scope of the Error	Description
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 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.

Table A.1 CUDA Exception Codes (continued)

Exception code	Precision of the Error	Scope of the Error	Description
CUDA_EXCEPTION_12: "Warp Assert"	Precise	Per warp	This occurs when any thread in the warp hits a device side assertion.
"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 B KNOWN ISSUES

The following are known issues with the current release.

- Kernel launches larger than 16MB are not currently supported by CUDA-MEMCHECK and may return erroneous results.
- Applications run much slower under CUDA-MEMCHECK. This may cause some kernel launches to fail with a launch timeout error when running with CUDA-MEMCHECK enabled.
- Without CUDA-MEMCHECK, when an application causes an access violation the kernel launch could fail with an error code of Unspecified Launch Failure.
- When using the **`--continue`** flag, CUDA-MEMCHECK tries to continue execution of the kernel and you may see more than one error being detected.
- Use of the **`--continue`** flag is not supported after a hardware exception has been received.
- On Mac OS X 10.6, the standalone CUDA-MEMCHECK tool will only run in blocking mode. Additionally, CUDA-MEMCHECK will run with continue mode enabled and will keep executing even after encountering the first error.
- On Windows XP, the standalone CUDA-MEMCHECK tool will only run in blocking mode.
- When running CUDA-MEMCHECK in integrated mode with CUDA-GDB, the following features are disabled:
  - Nonblocking launches
  - Device side allocation checking
  - Leak checking
  - Continue mode

Notice

Trademarks

Copyright