# CUDA, Supercomputing for the Masses: Part 16

CUDA 3.0 provides expanded capabilities

March 25, 2010
URL:http://www.drdobbs.com/parallel/cuda-supercomputing-for-the-masses-part/224200312

In CUDA, Supercomputing for the Masses: Part 15 of this article series, I discussed mixing CUDA and OpenGL within the same application by utilizing a PBO (Pixel Buffer Object) to create images with CUDA on a pixel-by-pixel basis and display them using OpenGL. In this article and the next, I discuss the CUDA 3.0 release. Far from an update to just support the newest 20-series architecture (codename "Fermi"), CUDA 3.0 is a major revision number increment release that adds enhancements valuable to all CUDA developers to make day-to-day development tasks easier, less error prone, and more consistent.

## Overview

Expanded consistent coverage appears to have been the thinking behind this major revision number release as it fills in several previous gaps and must-have capabilities. In a nutshell, there appear to be four main focus areas for the CUDA 3.0 release:

- Consistency and interoperability between the CUDA runtime and driver codes, OpenGL and DirectX, as well as versioning and significant additions to CUDA-GDB including a memory checker to find misaligned and out-of-bounds errors.
- C++ Class and Template inheritance.
- Increased OpenCL capability, which will not be covered in this article.
- Early Fermi architecture support, which will be covered in-depth in a separate article that will quickly follow this two-part series on the CUDA 3.0 release.

For many developers, installing CUDA 3.0 is a "must get" because this release supports CUDA driver and runtime buffer interoperability. In addition, CUDA-GDB can now debug driver API codes!

While this series has been focused on teaching the CUDA with the runtime API (e.g., those methods that start with "cuda" as opposed to "cu") because it is fairly intuitive and not too verbose, many developers utilize the driver API. As the CUDA Programmers Guide Chapter 3 notes:

> [T]he CUDA driver API requires more code, is harder to program and debug, but offers a better level of control and is language-independent since it handles binary or assembly code.

As a result, many -- especially commercial developers -- will be overjoyed because both debugging and interoperability enhancements permit better use of existing code bases and new development.

Similarly, those developers tasked with delivering production codes, or codes that need to run on a variety of systems without recompilation, will be very happy that the CUDA Toolkit libraries supports versioning. CUDA has evolved rapidly and has adding valuable new features with each release. Now an application can verify the driver and libraries on a system will provide the correct feature set. Similarly, multiple versions can be explicitly used just in case an older version -- perhaps to utilize a deprecated feature -- is required for a particular routine or to utilize a legacy library.

The CUDA 3.0 release will be a very popular download for C++ programmers because inheritance is now supported for both classes and templates! For non-C++ programmers, inheritance is a fundamental concept in the C++ language. Without a doubt, the CUDA 3.0 release will enable more C++ classes and libraries that can efficiently use both GPU and CPU resources. This in turn can open the doors to portable software and powerful hybrid computing models as well as facilitating porting efforts for existing C++ commercial applications to GPUs and other profit-based CUDA software development. C++ and CUDA support is a topic for an future article.

A new separate version of the CUDA C Runtime (CUDART) for emulation-mode debugging made it into this release. While worth mentioning, please note that emulation mode debugging has been deprecated as of the CUDA Toolkit 3.0 release and will be removed in a future release (forum link). Those who wish an emulated environment should look to a new project called gpuocelot lets CUDA programs run on x86 architecture CPUs without recompilation. More information can also be found in the paper, Translating GPU Binaries to Tiered SIMD Architectures with Ocelot. Ocelot also includes some debugging capabilities.

Be aware that you need to have the latest NVIDIA driver installed to use the CUDA 3.0 toolkit. As always, the latest released driver can be downloaded from CUDA Zone and installed for a number of systems. Beta drivers and software can be downloaded from nvdevelopers but registration is required. Ubuntu users might wish to follow the tutorial at Web Upd8 to see how to install the latest released or beta drivers via the Ubuntu tools.

## CUDA 3.0 Enhancements Affecting CUDA-GDB

CUDA 3.0 delivers a big performance improvement for programs compiled with debugging enabled (using `nvcc -g -G`) because register variables are no longer forced to be spilled to local memory. This alone can help speed the debugging workflow and make it easier to find bugs faster.

Another important enhancement is that CUDA 3.0 executables now use the standard ELF executable format. While opaque, this is actually a change that can affect many developers. For example, those who like to streamline their workflow by keeping CUDA-GDB running while changing and recompiling source will be happy to know that this now works in CUDA 3.0. Just as with GDB, CUDA-GDB does not need to be stopped and restarted every time the executable changes. If you don't use this feature then try it because it preserves breakpoints, debugging history and other information that can maintain the state of a debugging session and really cut down on retyping

In addition, developers can now look at register variables and examine the GPU memory space by casting pointers with CUDA-GDB. The values of "shadow" variables (variables with the same name inside an inner scope as shown for variable, as in the code snippet below) are also reported correctly.

```
{
    register int a=2;
   // do something with a
   {
       register int a=10;
       // do something with the value of a that is local to this scope
    }
}
```

In total, this latest version of CUDA-GDB provided in the CUDA 3.0 release delivers a significantly more consistent and seamless experience that exhibits fewer and fewer departures from standard or "expected" GDB behavior. Unquestionably, this uniformity will increase developer productivity, reduce frustration, and speed debugging efforts.

Be aware that some of the CUDA-GDB commands have changed since my Part 14 tutorial on CUDA-GDB. Specifically the commands `info cuda state`, and `info cuda threads all` have been removed. The new `info` commands are:

- info cuda system
- info cuda device
- info cuda sm
- info cuda warp
- info cuda lane

Also, there is the wonderful new command `set cuda memcheck on` to check for out-of-bounds and misaligned address errors! Later in this article, I explore the capabilities of the memory checker in CUDA-GDB. The memory checker can also be called separately with the standalone tool cuda-memcheck, which can help automate testing and support field debugging.

For additional information, the latest documentation on CUDA-GDB can be found in version 3.0 of the NVIDIA CUDA Debugger CUDA-GDB. The "What's New" section provides an overview of changes and the "Known Issues" section discusses important limitations. Here are three important ones that still exist:

- X11 cannot be running on the GPU that is used for debugging. Suggested workarounds include:
  - Remote access to a single GPU (VNC, ssh, etcetera).
  - Use two GPUs, where X11 is running on only one of the graphics processors.
- Multi-GPU applications are not supported.
- Kernel launches are not asynchronous as the driver enforces blocking kernel launches when debugging is enabled. This can slow application performance and affect asynchronous behavior.

The following simple example code, version.cu, demonstrates how to read the current driver and runtime version. Please note that the reported version numbers returned differ from what you might expect as discussed in the forums.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int driverVersion, runtimeVersion;
  cudaDriverGetVersion(&driverVersion);
  cudaRuntimeGetVersion(&runtimeVersion);
  printf("driver version %d runtime version %d\n",
         driverVersion, runtimeVersion);
  return 0;
}
```

## Using CUDA-GDB Enhancements

Let's put the following example code into a file called assign.cu. Please note that this code contains an out-of-bounds memory error because `cudaMalloc()` was called with the incorrect number of items: `(N-1)` rather than `N`.

```
#include <stdio.h>
#include <stdlib.h>

// Simple assignment test
#define N 256

__global__ void kernel(unsigned int *data, int n)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if(idx < n) data[idx] = idx;
}

int main(void)
{
    int i;
    unsigned int *d = NULL;
    unsigned int odata[N];

    cudaMalloc((void**)&d, sizeof(int) * (N-1) );

    kernel<<<1, N>>>(d,N);

    cudaMemcpy(odata, d, sizeof(int)*N, cudaMemcpyDeviceToHost);

    // Test to see if the array retrieved from the GPU is correct
    for (i = 0; i < N; i++) {
      if(odata[i] != i) {
        break;
      }
    }
    if(i == N) printf("PASSED\n");
    else printf("FAILED\n");

    cudaFree((void*)d);
    return 0;
}
```

The following command will build it:

```
nvcc -g -G assign.cu -o assign
```

Now start the executable under CUDA-GDB and run it. (Please note that X Windows cannot be running on the GPU that will be used to debug the code.) As can be seen, the program appears to run correctly because it indicates the golden test was passed.

```
$ cuda-gdb assign
NVIDIA (R) CUDA Debugger
BETA release
Portions Copyright (C) 2008,2009 NVIDIA Corporation
GNU gdb 6.6
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-unknown-linux-gnu"...
Using host libthread_db library "/lib/libthread_db.so.1".
(cuda-gdb) r
Starting program: PATHNAME/assign
[Thread debugging using libthread_db enabled]
[New process 3128]
[New Thread 140278242694928 (LWP 3128)]
PASSED

Program exited normally.
(cuda-gdb)
```

Now tell cuda-gdb to check memory accesses by typing the command:

```
set cuda memcheck on
```

Rerunning the program under CUDA-GDB now generates a segmentation fault, which indicates there is a memory error in the program.

```
(cuda-gdb) run
Starting program: PATHNAME/assign
[Thread debugging using libthread_db enabled]
```

```
[New process 3145]
[New Thread 140147342771984 (LWP 3145)]

Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 140147342771984 (LWP 3145)]
[Current CUDA Thread <<<(0,0),(224,0,0)>>>]
0x0000000000feed18 in kernel <<<gridDim=(1,1), blockDim=(256,1,1)>>> (
    data=0x101000, n=256) at assign.cu:10
10        if(idx < n) data[idx] = idx;
(cuda-gdb)
```

The CUDA-GDB line (underlined in the previous output) tells us that a thread in the range block 0,0 thread 224-256 manifests the problem.

The command `info cuda warp` verifies the number of threads in a warp with the underlined value of 32 below:

```
(cuda-gdb) info cuda device
DEV:  0/1    Device Type: gt200   SM Type: sm_13   SM/WP/LN: 30/32/32   Regs/LN: 128
 … more output
```

Examining the program should verify the issue is in allocating one too few integers. Double-check this by setting a breakpoint at line 10 in the CUDA kernel and rerun the program.

```
b 10
Breakpoint 1 at 0xfeea50: file assign.cu, line 10.
(cuda-gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
```

CUDA-GDB stops at the breakpoint.

```
Starting program: PATHNAME/assign
Breakpoint 1 at 0x410bfa: file assign.cu, line 10.
[Thread debugging using libthread_db enabled]
[New process 3597]
[New Thread 140121555752720 (LWP 3597)]
Breakpoint 1 at 0x1575a50: file assign.cu, line 10.
[Switching to Thread 140121555752720 (LWP 3597)]
[Current CUDA Thread <<<(0,0),(224,0,0)>>>]

Breakpoint 1, kernel <<<gridDim=(1,1), blockDim=(256,1,1)>>> (data=0x101000,
    n=256) at assign.cu:10
10        if(idx < n) data[idx] = idx;
```

Now repeatedly step through the program to find when the problem manifests itself. Recall that CUDA-GDB supports stepping GPU code only at the finest granularity of a warp. This means that typing `next` or `step` from the CUDA-GDB command line (when in the focus of device code) advances all threads in the same warp as the current thread of focus. (Multiple commands can be issued with `s #`, where # is some integer value.)

```
((cuda-gdb) s
[Current CUDA Thread <<<(0,0),(192,0,0)>>>]
kernel <<<gridDim=(1,1), blockDim=(256,1,1)>>> (data=0x101000, n=256)
    at assign.cu:11
11      }
(cuda-gdb) s
[Current CUDA Thread <<<(0,0),(0,0,0)>>>]
kernel <<<gridDim=(1,1), blockDim=(256,1,1)>>> (data=0x101000, n=256)
    at assign.cu:11
11      }
(cuda-gdb) s
[Current CUDA Thread <<<(0,0),(32,0,0)>>>]
kernel <<<gridDim=(1,1), blockDim=(256,1,1)>>> (data=0x101000, n=256)
    at assign.cu:11
11      }
(cuda-gdb) s
[Current CUDA Thread <<<(0,0),(64,0,0)>>>]
kernel <<<gridDim=(1,1), blockDim=(256,1,1)>>> (data=0x101000, n=256)
    at assign.cu:11
11      }
(cuda-gdb) s
[Current CUDA Thread <<<(0,0),(96,0,0)>>>]
kernel <<<gridDim=(1,1), blockDim=(256,1,1)>>> (data=0x101000, n=256)
    at assign.cu:11
11      }
(cuda-gdb) s
[Current CUDA Thread <<<(0,0),(128,0,0)>>>]
```

```
kernel <<<gridDim=(1,1), blockDim=(256,1,1)>>> (data=0x101000, n=256)
    at assign.cu:11
11      }
(cuda-gdb) s
[Current CUDA Thread <<<(0,0),(160,0,0)>>>]
kernel <<<gridDim=(1,1), blockDim=(256,1,1)>>> (data=0x101000, n=256)
    at assign.cu:11
11      }
(cuda-gdb) s
[Current CUDA Thread <<<(0,0),(224,0,0)>>>]
kernel <<<gridDim=(1,1), blockDim=(256,1,1)>>> (data=0x101000, n=256)
    at assign.cu:11
11      }
(cuda-gdb) s
0x00007fc0dee29eb7 in sched_yield () from /lib/libc.so.6
(cuda-gdb)
```

From the output, it is clear that a thread in the range 224 – 256 generates the memory fault.

If your Linux shell supports it, please suspend CUDA-GDB in the current window. Then modify the source to replace the `(N-1)` with `N` as shown in the corrected source below and rebuild the executable. (Generally, the editor is running in another window so it does not need to be stopped and restarted.)

```
__global__ void kernel(unsigned int *data, int n)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if(idx < n) data[idx] = idx;
}

int main(void)
{
    int i;
    unsigned int *d = NULL;
    unsigned int odata[N];

    cudaMalloc((void**)&d, sizeof(int) * N );

    kernel<<<1, N>>>(d,N);

    cudaMemcpy(odata, d, sizeof(int)*N, cudaMemcpyDeviceToHost);

    // Test to see if the array retrieved from the GPU is correct
    for (i = 0; i < N; i++) {
      if(odata[i] != i) {
        break;
      }
    }
    if(i == N) printf("PASSED\n");
    else printf("FAILED\n");

    cudaFree((void*)d);
    return 0;
}
```

Now restart CUDA-GDB and tell it to run the program with the `run` command. CUDA-GDB indicates that the program has changed as seen below:

```
run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
'PATHNAME/assign' has changed; re-reading symbols.
Breakpoint 1 at 0x410bfa: file assign.cu, line 10.
Starting program: PATHNAME/assign
[Thread debugging using libthread_db enabled]
[New process 4276]
[New Thread 139863222687504 (LWP 4276)]
Breakpoint 1 at 0x14fc730: file assign.cu, line 10.
[Switching to Thread 139863222687504 (LWP 4276)]
[Current CUDA Thread <<<(0,0),(0,0,0)>>>]

Breakpoint 1, kernel <<<gridDim=(1,1), blockDim=(256,1,1)>>> (data=0x210000, n=256) at assign.cu:10
10        if(idx < n) data[idx] = idx;
(cuda-gdb)
```

Note that the break point at line 10 in the source is still set, so type `continue` (abbreviated to `c` in the example below). CUDA-GDB indicates the program exited normally and the program itself indicates it passed the test.

```
c
Continuing.
```

```
PASSED

Program exited normally.
(cuda-gdb)
```

Finally, note that `cudaMalloc()` does not allocate in character (single-byte) sizes. For this reason, a common typographical error (e.g., forgetting the parenthesis by typing N-1) shown below will not be caught even through the incorrect number of bytes were specified in the allocation of `d`. Both our test program and CUDA-GDB will miss this error problem -- even when memory checking is enabled.

```
__global__ void kernel(unsigned int *data, int n)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if(idx < n) data[idx] = idx;
}

int main(void)
{
    int i;
    unsigned int *d = NULL;
    unsigned int odata[N];

    cudaMalloc((void**)&d, sizeof(int) * N -1 );

    kernel<<<1, N>>>(d,N);

    cudaMemcpy(odata, d, sizeof(int)*N, cudaMemcpyDeviceToHost);

    // Test to see if the array retrieved from the GPU is correct
    for (i = 0; i < N; i++) {
      if(odata[i] != i) {
        break;
      }
    }
    if(i == N) printf("PASSED\n");
    else printf("FAILED\n");

    cudaFree((void*)d);
    return 0;
}
```

Again, suspend CUDA-GDB, modify the `cudaMalloc()` line as shown above, rebuild the executable, and restart CUDA-GDB.

For convenience, delete breakpoint 1 and run the program. Note that the program indicates success and the debugging session completes without a segmentation fault even though memory checking is still enabled.

```
delete 1
(cuda-gdb) r
'PATHNAME/assign' has changed; re-reading symbols.
Starting program: PATHNAME/assign
[Thread debugging using libthread_db enabled]
[New process 3988]
[New Thread 139924018698000 (LWP 3988)]
PASSED

Program exited normally.
(cuda-gdb)
```

## Summary

CUDA 3.0 is a major revision number release that delivers important benefits for C++, OpenCL, CUDA driver API and CUDA runtime API developers that will likely make this a "must install" version. In addition, removal of interoperability barriers between driver and runtime API as well as DirectX and OpenGL creates new opportunities for code development and integration of existing software projects. The next article in this series will discuss how to utilize these expanded capabilities.

As noted, debugging has taken a big step forward in this release. CUDA-GDB in particular has been dramatically improved so that it provides a seamless and consistent debugging experience for both runtime and driver API developers. The addition of memory checking makes CUDA-GDB an even more powerful tool. The inclusion of the cuda-memcheck command-line utility can help automate testing and support field debugging.

## For More Information

- CUDA, Supercomputing for the Masses: Part 15
- CUDA, Supercomputing for the Masses: Part 14
- CUDA, Supercomputing for the Masses: Part 13
- CUDA, Supercomputing for the Masses: Part 12

- CUDA, Supercomputing for the Masses: Part 11
- CUDA, Supercomputing for the Masses: Part 10
- CUDA, Supercomputing for the Masses: Part 9
- CUDA, Supercomputing for the Masses: Part 8
- CUDA, Supercomputing for the Masses: Part 7
- CUDA, Supercomputing for the Masses: Part 6
- CUDA, Supercomputing for the Masses: Part 5
- CUDA, Supercomputing for the Masses: Part 4
- CUDA, Supercomputing for the Masses: Part 3
- CUDA, Supercomputing for the Masses: Part 2
- CUDA, Supercomputing for the Masses: Part 1

---

*Rob Farber is a senior scientist at Pacific Northwest National Laboratory. He has worked in massively parallel computing at several national laboratories and as co-founder of several startups. He can be reached at rmfarber@gmail.com.*