CUDA, Supercomputing for the Masses: Part 14

Debugging CUDA and using CUDA-GDB

By Rob Farber , Dr. Dobb's Journal
Oct 16, 2009
URL:http://www.ddj.com/hpc-high-performance-computing/220601124

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.

In CUDA, Supercomputing for the Masses: Part 13 of this article series, I resumed the discussion of "texture memory" began in Part 11 of this series and included information such as the recently introduced CUDA Toolkit 2.2 that added ability to write to global memory on the GPU that has a 2D texture bound to it. This installment focuses on debugging techniques and how CUDA-GDB can be used to effectively diagnose and debug CUDA code -- with an emphasis on how to speed the process when looking through large amounts of data as well as the thread syntax and semantic differences needed to debug kernels while they are running on the GPU.

Upfront, youo need to be aware of two important points about the current state of CUDA-GDB:

- It only runs on UNIX-based systems. Microsoft users in particular should look to the recently announced Visual Studio debugger, which I'll cover in a future column.
- X11 cannot be running on the GPU that is used for debugging. Instead use either a multi-GPU system or kill X11 and remotely access the single-GPU system via ssh, VNC, or some other method.

This article also provides an admittedly contrived debugging example. I recommend everyone look though it -- regardless of their previous experience with GDB -- for the following reasons:

- Everyone new to CUDA-GDB: Example commands demonstrate how to use the new CUDA thread syntax and highlight semantic changes that allow debugging and stepping through kernels running on CUDA-enabled graphics processors.
- Beginners: This example shows how to set breakpoints (both symbolically and by line number) as well as performing other basic debugging operations including: running the program, stepping execution through the source code line-by-line, continuing program execution, and exiting the debugger.
- Experts: GPUs are fast, which generally means that debugging a CUDA application requires digging through large amounts of data to find a problem. Putting a little thought into debugging strategies including the application of debugger helper functions and GDB artificial arrays can really save debugging time (and personal sanity).

Essentially CUDA-GDB is a port of the GNU GDB debugger version 6.6. Programmers who have used GDB in the past will find they are already familiar with CUDA-GDB and should look to this article for CUDA specific tips. Newcomers to GDB will be able to use this article to begin debugging their software right away, but should look to one of the many extensive tutorials and references on the Internet to find out more about the comprehensive set of features available within this powerful debugging tool. One obvious starting place is the GNU documentation for GDB. Regardless of skill level, all CUDA developers should at least glance over the latest version of CUDA-GDB: The NVIDIA CUDA Debugger.

Debugging Methods Prior to CUDA-GDB

Prior to the creation of CUDA-GDB, the easiest and least elegant method of debugging a CUDA program was to add print statements to the source code and compile to run in the emulator (initiated by passing the **-deviceemu** or **-device-emulation** flag to the nvcc compiler). Since the emulator runs on the host processor (and not on the GPU), the **print** statements can be compiled and linked so the programmer can examine whatever program values might be important. This awkward method was one of the most unobtrusive ways to see what was going on inside a CUDA program in the early days of CUDA. (This debugging method was discussed way back in Part 1 of this series in April 2008.) I mention this method again because it might -- as a method of last resort -- help someone find bugs in their code. Basically, if you don't trust what the GPU is doing -- try running on the emulator. If the code still fails, you know it is not the GPU. Keep in mind that the emulator does not precisely reproduce what happens on the GPU, which means that bugs and behavior that occur on the GPU (including race conditions) may not happen in the emulated environment.

An alternative method that actually utilizes the GPU and permits examination of GPU calculated results utilizes **cudaMemcpy()** to transfer any variables of interest from the GPU to a scratch location on the host. Host-based methods (including GDB and/or print statements) can

then be used to examine the information in the scratch location to hopefully diagnose the problem. Later in this article, I will use this technique to demonstrate a simple helper debugging function for CUDA-GDB to identify an error when writing to a large CUDA vector.

Mapped Memory and the Importance of Regression Testing

It is worthwhile mentioning at this point that the new mapped memory capability, introduced in the CUDA 2.2 release, provides an important (and convenient) new capability to facilitate regression testing when porting legacy application code to CUDA. Without question, regression testing is an essential software practice. It cannot be emphasized too strongly how important this technique is in creating and verifying correctly working software!

In the case of legacy software, the developer already has a working code base that can be used for comparison with GPU generated results to help identify errors. Mapped memory (discussed in Part 12, "CUDA 2.2 Changes the Data Movement Paradigm") greatly facilitates this process by transparently maintaining a synchronized version of data between both the host and device memory spaces. With care, the programmer can exploit this transparent synchronization to keep the original software functional throughout the entire porting project. As a result, there will be a known working version that can be used to compare all GPU results and intermediate results.

Essentially new CUDA kernels can be incorporated into the legacy code without having to think about explicitly moving data off the host and onto the GPU, which allows easy switching between the new CUDA kernel(s) and corresponding original host code. The new GPU version can then be evaluated on one or more test cases to see if it produces correct results. If an error is identified, then the original host code for that phase of the calculation -- plus intermediate results -- can be used to quickly identify the first appearance of the error in the GPU code. Eventually, enough of the calculation will reside on the GPU so it no longer becomes necessary to maintain synchronization with the host and mapping can be disabled or removed -- thus creating a GPU only version of the legacy code that can run a full-speed without any PCI bottlenecks.

Tips for Using CUDA-GDB with Large Data

Graphics processors are excellent platforms for working with large amounts of data due to their many-core architecture, massive threading model, and high performance. Conversely, finding errors in these large volumes of data by manual methods can be time consuming and painful.

To make debugging easier, I like to include some simple helper debugging routines in my code that can be called from GDB, CUDA-GDB, or used as part of a test harness for sanity checking through the use of assertions. When running in production, these debugging routines are never called (if necessary they can be eliminated with **#ifdef** statements) so they do not incur any memory or processor overhead. Having the ability to interactively call any of a number of debugging routines when running GDB (or CUDA-GDB) provides a convenient and easy way to look for errors while the program is executing -- and without having to modify the source code or recompile!

The function **sumOnHost()** in the AssignScaleVectorWithError.cu example program below provides one illustration of a helpful debugging routine. In this case, **sumOnHost()** calculates the floating-point sum of a large vector after moving the vector from device to host. It is easy to imagine how the same idea can be extended to provide useful information about any large data structure. Calculating a sum is useful because it produces a single number that can be used to get a sense of the data, identify NaN (Not a Number) problems and perform other sanity checks. Many network and disk subsystems use a similar technique by calculating a checksum (or other aggregate measure) to look for data errors.

Using a sum as a comparative measure to spot data differences can be especially useful when known-working host-based software exists that can be used to compare intermediate results against a CUDA-based kernel. Instead of blindly looking through huge tables of numbers to find a difference, the programmer can leverage the power of CUDA-GDB to quickly isolate the first occurrence of any differences between the legacy host and GPU kernel results and/or intermediate values.

From experience, this is a wonderful time-saving capability. Be aware that some variability will inevitably occur when comparing floating-point results because floating-point is only an approximate representation. Minor differences in how the host or GPU performs arithmetic and even legitimate variations in the ordering of arithmetic operations (which can caused by simple changes like using different compiler switches or changing optimization levels) can cause slight variations in the results of even correctly working code.

Most Unix-based operating system releases of the CUDA Toolkit now include CUDA-GDB, so you should be able to just type:

```
cuda-gdb
```

to start the debugger. (If not, look to the CUDA-GDB release notes and the NVIDIA CUDA forums to see how others have gotten the debugger to work on your specific OS.)

CUDA-GDB accepts the same variety of arguments and options as GDB. Usually CUDA-GDB is started with one argument that specifies the program executable to debug (e.g., cuda-gdb a.out). CUDA-GDB can also be used to debug programs that are already running by adding the process ID of the program (PID) to the command-line (e.g., cuda-gdb a.out pid).

To debug programs in a human-friendly fashion, the compiler needs to generate additional debugging information for CUDA-GDB that describes the data type of each variable or function and the correspondence between source line numbers and addresses in the executable

code. To make the compiler generate this information, both the **-g** and **-G** options must be specified when you run the nvcc compiler.

The following is the command line to compile the program AssignScaleVectorWithError.cu for debugging:

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

So, what do these command-line options do?

- The **-G** options specifies generate debugging information for the CUDA kernels and
  - Forces **-O0** (mostly unoptimized) compilation
  - Spills all variables to local memory (and will probably slow program execution)
- The **-g** option tells nvcc to generate debugging information for the host code and include symbolic debugging information in the executable.
- Finally, the **-o** option tells the compiler to write the executable to AssignScaleVectorWithError.

NOTE: It is currently not possible to generate debugging information when compiling with the **-cubin** option.

The following is the source code for AssignScaleVectorWithError.cu:

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

// A simple example program to illustrate
// debugging with cuda-gdb

// Vector on the device
float *a_d;

// Print a message if a CUDA error occurred
void checkCUDAError(const char *msg) {
  cudaError_t err = cudaGetLastError();
  if( cudaSuccess != err) {
    fprintf(stderr, "Cuda error: %s: %s.\n", msg, cudaGetErrorString( err) );
    exit(EXIT_FAILURE);
  }
}

// Zero the vector
__global__ void zero(float *v_d, int n)
{
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  if(tid < n)
    v_d[tid] = 0.f;
}

// Assign the vector with consecutive values starting with zero
__global__ void assign(float *v_d, int n)
{
  int tid = threadIdx.x;
  if(tid < n)
    v_d[tid] = ((float) tid);
}

// Scale the vector
__global__ void scale(float *v_d, int n, float scaleFactor)
{
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  if(tid < n)
    v_d[tid] *= scaleFactor;
}

// Move the vector to the host and sum
float sumOnHost(const float *v_d, int n)
{
  float sum=0.f;
  int i;


  // create space on the host for the device data
  float *v_h = (float*)malloc(n*sizeof(float));

  // check if the malloc succeeded
  assert(v_h != NULL);

  // copy the vector from the device to the host
```

```
  cudaMemcpy(v_h,v_d, n*sizeof(float), cudaMemcpyDeviceToHost);

  for(i=0; i<n; i++) sum += v_h[i];

  // free the vector on host
  free(v_h);

  return(sum);
}

int main()
{
  int nBlocks = 32;
  int blockSize = 256;
  int n = nBlocks*blockSize;
  float scaleFactor = 10.f;

  // create the vector a_d on the device and zero it
  cudaMalloc((void**)&a_d, n*sizeof(float));
  checkCUDAError("Create and zero vector");

  // fill the vector with zeros
  zero<<<nBlocks, blockSize>>>(a_d, n);
  // assign the vector
  assign<<<nBlocks, blockSize>>>(a_d, n);
  // scale the vector by scaleFactor
  scale<<<nBlocks, blockSize>>>(a_d, n, scaleFactor);

  // calculate the sum of the vector on the host
  float dSum = sumOnHost(a_d, n);
  checkCUDAError("calculating dSum");

  // Check if both host and GPU agree on the result
  float hSum=0.f;
  for(int i=0; i < n; i++) hSum += ((float)i)*scaleFactor;

  if(hSum != dSum) {
    printf("TEST FAILED!\n");
  } else {
    printf("test succeeded!\n");
  }

  // free the vector on the device
  cudaFree(a_d);
}
```

In a nutshell, this program creates a vector on the device, **a_d**, which is filled with zeros by the kernel, **zero()**. The vector **a_d** is then assigned consecutively increasing values, starting at zero by the kernel, **assign()**. Finally, the vector **a_d** is multiplied by a scale factor with kernel, **scale()**. The host routine **sumOnHost()** is called to calculate the sum of the values in **a_d**, which is placed in **dSum** and compared against the host generated sum contained in the variable **hSum**. If the values are the same, we get a message that states the test succeeded. Otherwise, the program indicates the test failed.

As we see below, running the unmodified program generates a failure message, which indicates there is a bug in the code:

```
$ ./AssignScaleVectorWithError
TEST FAILED!
```

The following command starts CUDA-GDB so it can be used to debug the program:

```
$cuda-gdb AssignScaleVectorWithError
```

You should see output similar to the following:

```
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".
```

We use the abbreviated command **l** (for list) to look at the lines around line 81 in the source code:

```
(cuda-gdb) l 81
76          checkCUDAError("Create and zero vector");
77
78          // fill the vector with zeros
79          zero<<<nBlocks, blockSize>>>(a_d, n);
80          // assign the vector
81          assign<<<nBlocks, blockSize>>>(a_d, n);
82          // scale the vector by scaleFactor
83          scale<<<nBlocks, blockSize>>>(a_d, n, scaleFactor);
84
85          // calculate the sum of the vector on the host
```

Now we set a breakpoint prior to starting execution of the **assign()** kernel at line 81 with the command (again using the one letter abbreviation **b** for "breakpoint"):

```
(cuda-gdb) b 81
Breakpoint 1 at 0x40f216: file AssignScaleVectorWithError.cu, line 81.
```

Breakpoints can also be set symbolically as shown with the following command that sets a breakpoint whenever the kernel **scale()** is called:

```
(cuda-gdb) b scale
Breakpoint 2 at 0x40f4e3: file AssignScaleVectorWithError.cu, line 38.
```

We now run the program in the debugger using the letter **r** instead of typing out the full command **run**. (Note: some of the output may appear different such as the process ID):

```
(cuda-gdb) r
Starting program: /home/XXX/DDJ/Part14/AssignScaleVectorWithError
[Thread debugging using libthread_db enabled]
[New process 16805]
[New Thread 140439601190656 (LWP 16805)]
[Switching to Thread 140439601190656 (LWP 16805)]

Breakpoint 1, main () at AssignScaleVectorWithError.cu:81
81          assign<<<nBlocks, blockSize>>>(a_d, n);
Current language:  auto; currently c++
```

Using the **p** command (for "print") we call the host function, **sumOnHost()**, with arguments appropriate to move all the data in the GPU array **a_d** to the host and calculate a sum of the values. As can be seen, the call to the kernel **zero()** appears to have worked correctly as the vector seems to be filled with only zero floating-point values:

```
 (cuda-gdb) p sumOnHost(a_d, n)
$1 = 0
```

We use the **next** command (abbreviated **n**) to run the next line of the program. In this case, the program runs the **assign()** kernel on the GPU.

Please note that unlike normal execution, calls to a kernel in CUDA-GDB happen synchronously. (Normally kernels are launched asynchronously).

Thus after typing the **next** command, control returns only after the **assign()** kernel runs to completion on the GPU.

As pointed out in Section 4.4 of the CUDA-GDB manual, the debugger support stepping GPU code at the granularity of a warp. This means that individual threads are not advanced but rather that all the threads within the warp advance. The exception is stepping over a thread barrier call, **__syncThreads()**, which causes all the threads to advance past the barrier. Additionally, it is not possible to step over a subroutine because the compiler currently inlines this code. Thus, it is only possible to step into a subroutine.

Again we look at the sum that is returned from **sumOnHost()** with the print command:

```
(cuda-gdb) n
83        scale<<<nBlocks, blockSize>>>(a_d, n, scaleFactor);
(cuda-gdb) p sumOnHost(a_d, n)
$2 = 32640
```

In this case the returned value of **32640** looks suspiciously small to be the sum of all integers ranging from [0 to **nBlocks*BlockSize**), so we elect to "continue" (abbreviated with **c**) until we hit the next breakpoint that happens to be the breakpoint set in the CUDA kernel **scale()**. (Note: For the moment, we ignore the meaning of the line that describes the "Current CUDA Thread".)

```
(cuda-gdb) c
Continuing.
[Current CUDA Thread <<<(0,0),(0,0,0)>>>]

Breakpoint 2, scale () at AssignScaleVectorWithError.cu:38
38        int tid = blockIdx.x * blockDim.x + threadIdx.x;
```

The debugger allows us to examine values on the GPU itself when in a kernel running on the GPU. This is the case as the breakpoint was set to stop program when inside the **scale()** kernel.

The address of **a_d** was passed into this kernel via the **v_d** function argument. Using the print command (abbreviated **p**), we can examine successive values of the vector values residing in the GPU memory by using the GNU concept of an artificial array. As can be seen in the output of the following command, the first 10 values (indicated by the syntax @10 in the command) of the vector were set correctly by the **assign()** kernel:

```
 (cuda-gdb) p *v_d@10
$3 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

However, we see that vector elements greater than 255 are still set to zero, which indicates there is a problem in the **assign()** kernel. This is accomplished by telling the debugger to offset 250 elements from the start of the **v_d** pointer with the command syntax **(v_d+250)**.

```
 (cuda-gdb) p *(v_d+250)@10
$4 = {250, 251, 252, 253, 254, 255, 0, 0, 0, 0}
```

We type **quit** to exit CUDA-GDB and go to look at our code:

```
 (cuda-gdb) quit
The program is running.  Exit anyway? (y or n) y
rmfarber@k2:~/DDJ/Part14$
```

It turns out that the problem with **assign()** is that the variable tid is only set to **threadIdx.x**. This is incorrect when multiple blocks are used.

Let's verify this by using the CUDA-GDB extensions that allow us to look at individual threads within a warp.

Start CUDA-GDB again:

```
$cuda-gdb AssignScaleVectorWithError
```

Now, let's set a breakpoint at line 31, which is after **tid** is initialized in the **assign()** kernel and run the debugger until the breakpoint is hit:

```
(cuda-gdb) b 31
Breakpoint 1 at 0x40f4cf: file AssignScaleVectorWithError.cu, line 31.
(cuda-gdb) r
Starting program: /home/XXXX/DDJ/Part14/AssignScaleVectorWithError
[Thread debugging using libthread_db enabled]
[New process 22405]
[New Thread 139839913080576 (LWP 22405)]
[Switching to Thread 139839913080576 (LWP 22405)]
[Current CUDA Thread <<<(0,0),(0,0,0)>>>]

Breakpoint 1, assign () at AssignScaleVectorWithError.cu:31
31        if(tid < n)
```

```
Current language:  auto; currently c++
```

Using the "info cuda threads" command, we see the following output:

```
(cuda-gdb) info cuda threads
<<<(0,0),(0,0,0)>>> ... <<<(31,0),(255,0,0)>>> assign ()
    at AssignScaleVectorWithError.cu:31
```

CUDA thread information is represented in the following form:

```
<<<(BX,BY),(TX,TY,TZ)>>>
```

Where **BX** and **BY** are the **X** and **Y** block indexes and **TX**, **TY**, and **TZ** are the corresponding thread **X**, **Y**, and **Z** indexes. Thus we can see that the **assign()** kernel has blocks with indexes ranging from (0,0) to (31,0) and threads within each block ranging from (0,0,0) to (255,0,0). This correctly represents a kernel configured to run on the GPU with 32 blocks where each block contains 256 threads per block.

The following line indicates the debugger is currently set to examine the first thread of the first block:

```
[Current CUDA Thread <<<(0,0),(0,0,0)>>>]
```

Printing **tid** shows that it is correctly set to zero for this thread"

```
(cuda-gdb) p tid
$1 = 0
```

Using the CUDA thread syntax, we switch to block 31 and thread 255 using an abbreviated syntax to save typing:

```
(cuda-gdb) thread <<<(31),(255)>>>
Switching to <<<(31,0),(255,0,0)>>> assign ()
    at AssignScaleVectorWithError.cu:31
31        if(tid < n)
```

Printing the value of the **tid** variable shows that it is incorrectly set to **255**.

```
(cuda-gdb) p tid
$2 = 255
```

We now know the **assign()** kernel incorrectly assigns **threadIdx.x** to **tid** with the following statement:

```
int tid = threadIdx.x;
```

Using an editor, change the assignment of the **tid** index to the following:

```
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
```

After saving, recompiling and running the modified program, we see that the program now reports success.

```
test succeeded!
```

Starting CUDA-GDB and repeating the previous debugging steps, we now see that **tid** in thread <<<(31,0),(255,0,0)>>> correctly contains the value of **8191**:

```
(cuda-gdb) thread <<<(31),(255)>>>
```

```
Switching to <<<(31,0),(255,0,0)>>> assign () at AssignVector.cu:31
31        if(tid < n)
(cuda-gdb) p tid
$1 = 8191
```

Additional CUDA-GDB Debugging Extensions and Semantics

CUDA-GDB provides a number of CUDA-specific commands:

- **thread** - Display the current host and CUDA thread of focus
- **thread <<<(TX,TY,TZ)>>>** - Switch to the CUDA thread at the specified coordinates
- **thread <<<(BX,BY),(TX,TY,TZ)>>>** - Switch to the CUDA block and thread at the specified coordinates
- **info cuda threads** - Display an overall summary of all CUDA threads that are currently resident on the GPU
- **info cuda threads all** - Display a list of each CUDA thread that is currently resident on the GPU. This can be quite large
- **info cuda state** - Display information about the current CUDA state

Special semantics of the next and step commands:

- Execution is advanced at the warp-level; all threads in the same warp as the current CUDA thread will proceed
- A special case is stepping the thread barrier call, **__syncthreads()**, which causes an implicit breakpoint to set immediately after the barrier. All threads are continued to this breakpoint after the **__syncthreads()**

Concerns and Known Issues

Please be aware of the following concerns and known issues as of the CUDA 2.3 release:

- Word size of the host system is no longer a concern as CUDA-GDB (as of the CUDA 2.2 Beta release) now supports both 32- and 64-bit systems
- There are reports in the forums that CUDA-GDB will sometimes hang. Be aware that this is a complex port of GDB and that NVIDIA appears to be doing a good job in fixing problems
- Any of the following might affect program behavior or performance when using the debugger:
  - X11 cannot be running on the GPU that is used for debugging. Suggested workarounds include:
    - Remote access to a single GPU (VNC, ssh, etc.)
    - Use two GPUs, where X11 is running on only one of the graphics processors
  - As of CUDA 2.2, the CUDA driver will automatically exclude the device running X11 from being picked by the application being debugged
  - Compiling with the **-G** option causes variables to be spilled to local memory, which can significantly reduce program performance. (As noted in Part 5, local memory can be up to 150x slower than register or shared-memory)
  - Kernel launches are no longer asynchronous as the debugger enforces blocking kernel launches
- Scope shadowing is not supported. This means that if a variable is introduced in an inner scope that has the same name as a variable in the outer scope, only the outer scope's value can be seen. The AssignArray.cu example demonstrates this restriction
- The debugger must be stopped in the kernel to examine device memory (allocated via **cudaMalloc**()) as device memory is not visible outside of the kernel function.
- Host memory that was allocated with **cudaMallocHost()** is not visible in CUDA-GDB
- Multi-GPU applications are not supported
- Not all illegal program behavior can be caught in the debugger, such as out-of-bounds memory accesses or divide-by-zero situations.
- It is not currently possible to step over a subroutine in device code
- These columns focus on using the runtime interface. Any programs using the device driver API cannot be debugged with CUDA-GDB because the device driver API is not supported

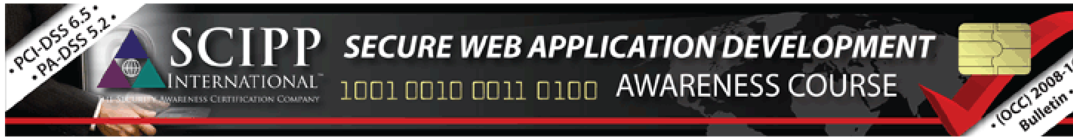
Summary

Proven software development strategies such as assertions and regression testing will greatly assist in developing software that is correct and bug free. When porting legacy software, look into the useful mapped memory features that are now available in CUDA. When bugs manifest themselves, CUDA-GDB can be used to track them down. Since GPU problems generally manipulate large amounts of data, artificial array and having a few simple routines in your code that can interactively be called from CUDA-GDB can really speed the debugging process.

For More Information

- 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



Copyright © 2009 United Business Media LLC