
CS264 Labs Documentation

Release 1

Justin Riley

February 03, 2011

CONTENTS

1	Welcome to CS264 Lab 1	3
1.1	Office Hours	3
1.2	Homework 0 Survey	3
1.3	Resonance GPU Cluster	3
1.4	Sign-up for an Amazon Web Services Account	3
1.5	Overview of CUDA Compiler Suite	4
1.6	Installing CUDA [optional]	4
1.7	Debugging CUDA using cuda-gdb	4
1.8	Using CUDA Memcheck to Detect Memory-Access Errors	8
1.9	Profiling CUDA code using the Visual Profiler	9
1.10	Source Code Control (GIT)	9
2	Welcome to CS264 Lab 2	11
2.1	Office Hours	11
2.2	Downloading the Lab Slides	11
2.3	Sign-up for an Amazon Web Services Account	11
2.4	Homework 0 Survey	11
2.5	Resonance GPU Cluster	11
2.6	Using Graphical Applications on the Resonance GPU Cluster	12
2.7	Debugging CUDA using cuda-ddd on Resonance	13
2.8	Using CUDA Memcheck to Detect Memory-Access Errors	26
2.9	Profiling CUDA code using the Visual Profiler	27
2.10	Source Code Control Using GIT	27
3	Indices and tables	31

Contents:

WELCOME TO CS264 LAB 1

Justin Riley (staff 'at' cs264 'dot' org)
Software Tools for Academics and Researchers
Office of Educational Innovation and Technology
Massachusetts Institute of Technology

1.1 Office Hours

Date/Time: Fridays 3:00pm - 5:00pm

Location: NE-48-308, 77 Massachusetts Ave, Cambridge, MA 02139

1.2 Homework 0 Survey

As part of homework 0 you are responsible for setting up all of the necessary user accounts in order to fully take advantage of the class. In order to fully complete homework 0 you *must* fill out the [homework 0 survey](#). The survey will be at the very bottom of Homework 0.

1.3 Resonance GPU Cluster

If you have not done so already you will need to create a SEAS account and request access to the resonance GPU cluster at Harvard. If you have issues with this please send an email to ircshelp@seas.harvard.edu. If you've just requested an account please wait at least a day before contacting the IRCS IT staff.

Once you have access to resonance, please use the [SSH Access to SEAS Hosts](#) guide to create an SSH key and add your new key to your `~/.authorized_keys` file on resonance.

1.4 Sign-up for an Amazon Web Services Account

NOTE: You will need a personal credit card in order to sign up for an Amazon Web Services Account.

To sign up for an Amazon Web Services account:

1. Sign up for AWS using either your existing Amazon account (if you have one) or by creating a new separate AWS account

2. After you've created your AWS account sign up for the EC2 web service. Use your AWS account email and password when prompted. Signing up for EC2 will automatically add you to the S3 web service which is required by EC2.
3. Next to obtain your AWS user ID navigate to your "Security Credentials" page.
4. Scroll down to the bottom of the page and look for the "Account Identifiers" section. You should see your "AWS Account ID". Please save your AWS account id somewhere safe.
5. After you've retrieved your AWS account ID please complete the HW0 survey in order to receive your \$200 worth of credit offered by Amazon for the course.

In order to familiarize yourself with the Amazon Web Services console please follow the instructions in Homework 0 to create a new SSH key for Amazon EC2.

1.5 Overview of CUDA Compiler Suite

The CUDA compiler suite consists of the following core tools:

- CUDA Compiler (nvcc) - converts your CUDA code to binary form
- CUDA Debugger (cuda-gdb) - allows you to "pause" and inspect your code while it's executing
- CUDA Memcheck (cuda-memcheck) - detects and reports memory access errors in your code
- CUDA Visual Profiler (computeprof) - used to measure performance and find potential opportunities for optimization
- and more...

1.6 Installing CUDA [optional]

You will be able to develop your code on the SEAS GPU cluster (resonance) or at the 53 Church St. computing labs, so no specific hardware is required for the course.

However, if you want to run your CUDA code on your own machine here's what you will need:

- A (fairly) recent NVIDIA graphics card (see [here](#) for a list of supported devices).
- CUDA 3.2 Toolkit
- CUDA 3.2 SDK

You will need to [download CUDA 3.2 Toolkit/SDK](#) for your OS and follow the install instructions provided by NVIDIA.

1.7 Debugging CUDA using cuda-gdb

When you're developing CUDA code there will be bugs and errors that can be difficult to figure out just by compiling and running the code alone. This is especially true when the problem lies within the GPU kernel itself. This is because a GPU kernel runs exclusively on the GPU and on a GPU "print statements" have no meaning or effect. Fortunately the CUDA toolkit comes with a CUDA debugger that allows you to "step" through your code and inspect local variables both in CPU code and within your GPU kernel(s). The debugger also allows you to switch to different threads on the GPU and inspect the local variable stack of a given GPU thread.

In order to debug CUDA you must compile your code with the debug flags enabled. For example:


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

The `-g` flag means compile the CPU portion of the code with debugging enabled. The `-G` flag means compile the GPU portion of the code with debugging enabled. These options are needed so that you can debug your code on both the CPU and GPU.

Let's try compiling the `example.cu` code included with HW0 on the resonance GPU cluster with the debug flags enabled:

```
nvcc example.cu -o example -I$CUDA_SDK_HOME/C/common/inc -L$CUDA_SDK_HOME/C/lib -lcutil_x86_64
```

This will create an executable or "binary" called `example` in the current directory.

Now that we've compiled the example CUDA code from Homework 0 we'll now use `cuda-gdb` to debug our code. Before we start here's a list of all the commands we'll use and some brief descriptions:

1. `breakpoint (b)` - set a "breakpoint" at key places in the code. the argument can either be a method name or line number
2. `run (r)` - run your application within the debugger. this must be executed before the rest of these commands can be used.
3. `next (n)` - move to the next line of code
4. `continue (c)` - continue to the next breakpoint or until program ends
5. `backtrace (bt)` - shows you where you are in the code
6. `thread` - lists the current CPU thread
7. `cuda thread` - lists the current active GPU threads (if any)
8. `cuda kernel` - lists the currently active GPU kernels and also allows you to switch 'focus' to a given GPU thread.

These commands are used throughout the following example.

Next we run the example code using the CUDA debugger. To do this we use the "`cuda-gdb`" command:

```
cuda-gdb --args example -string="tester"
NVIDIA (R) CUDA Debugger
3.2 release
Portions Copyright (C) 2008-2010 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 "/lib64/libthread_db.so.1".
```

This will bring you to a `cuda-gdb` prompt:

```
(cuda-gdb)
```

Next let's add some breakpoints. Breakpoints define places in the code to stop execution and allow the debugger to inspect the current state of threads and variables. In the example file we have `main`, `mangleCPU`, and `mangleGPU` methods. We will add a breakpoint at each method and run through 'stepping' through each line of these methods and inspecting local/global variables on different GPU threads. To set a breakpoint at each of these methods:

```
(cuda-gdb) b main
Breakpoint 1 at 0x4053fa: file example.cu, line 43.
(cuda-gdb) b mangleCPU
```

```
Breakpoint 2 at 0x403f1e: file example.cu, line 105.
(cuda-gdb) b mangleGPU
Breakpoint 3 at 0x4041fb: file example.cu, line 98.
```

Now that we have the breakpoints set, the debugger will stop at each method and allow us to inspect the method's local variables. In order to reach a breakpoint we must first run the code:

```
(cuda-gdb) r
Starting program: /home/jtriley/cudadb/example -string=tester
[Thread debugging using libthread_db enabled]
[New process 1072]
[New Thread 47173892559264 (LWP 1072)]
[Switching to Thread 47173892559264 (LWP 1072)]

Breakpoint 1, main (argc=2, argv=0x7fff3f6d5448) at example.cu:43
43      CUT_DEVICE_INIT(argc,argv);
```

Once the program starts it will run until a breakpoint is reached. In the above output we see that we're at Breakpoint 1 which stops at the main() method in our example.cu code. At this point let's inspect the CPU and GPU threads:

```
(cuda-gdb) thread
[Current thread is 2 (Thread 47173892559264 (LWP 1072))]
(cuda-gdb) cuda thread
No CUDA kernel is currently running.
```

Here the "thread" command displays information on the current CPU thread and "cuda thread" shows the current CUDA thread we're inspecting. Since we haven't yet reached the mangleGPU kernel breakpoint the command "cuda thread" just prints a message that no CUDA kernel is running.

For now let's skip past the main method and move to the mangleGPU method. We can allow the code to continue running until the next breakpoint using:

```
(cuda-gdb) c
Continuing.
Using device 0: Tesla T10 Processor
[Launch of CUDA Kernel 0 (mangleGPU) on Device 0]
[Switching to CUDA Kernel 0 (<<<(0,0),(0,0,0)>>>)]

Breakpoint 3, mangleGPU<<<(1,1),(6,1,1)>>> (instr=0x100000 <Address 0x100000 out of bounds>, outstr=0)
at example.cu:99
99      int i = blockIdx.x * blockDim.x + threadIdx.x;
```

Now we see that we're at the third breakpoint: mangleGPU. Notice that we did not first go to Breakpoint 2 (mangleCPU). This is because the order that the breakpoints will occur depends on what gets executed first by the running program. In this case mangleGPU is called before mangleCPU so the mangleGPU breakpoint occurs first.

Now that we're at the mangleGPU breakpoint, you'll notice that the "cuda thread" command now lists a CUDA kernel thread. Let's switch 'focus' to that thread:

```
(cuda-gdb) cuda thread
[Current CUDA kernel 0 (thread (0,0,0))]
(cuda-gdb) cuda kernel 0
[Switching to CUDA Kernel 0 (<<<(0,0),(0,0,0)>>>)]
99      int i = blockIdx.x * blockDim.x + threadIdx.x;
```

Next we'll inspect some local variables on the GPU (blockIdx, threadIdx):

```
(cuda-gdb) p blockIdx
$2 = {x = 0, y = 0}
(cuda-gdb) p threadIdx
```

```
$3 = {x = 0, y = 0, z = 0}
(cuda-gdb) p i
warning: Variable is not live at this point. Returning garbage value.
```

Notice that we can't yet access the integer "i" that gets created in the first line of the kernel. This is because we haven't yet stepped past this line. Let's step past:

```
(cuda-gdb) n
100     MANGLE(instr,outstr,i,len,x);
```

Now that we're at the next line after 'i' has been defined in the kernel we can inspect the value of 'i':

```
(cuda-gdb) p i
$4 = 0
```

Similarly we can switch to other GPU threads and check their value for 'i':

```
(cuda-gdb) cuda thread 1
[Switching to CUDA Kernel 0 (device 0, sm 0, warp 0, lane 1, grid 1, block (0,0), thread (1,0,0))]
100     MANGLE(instr,outstr,i,len,x);
(cuda-gdb) p i
$5 = 1
(cuda-gdb) cuda thread 2
[Switching to CUDA Kernel 0 (device 0, sm 0, warp 0, lane 2, grid 1, block (0,0), thread (2,0,0))]
100     MANGLE(instr,outstr,i,len,x);
(cuda-gdb) p i
$6 = 2
```

After you're done inspecting "i" across GPU threads let's continue on to the next breakpoint:

```
(cuda-gdb) c
Continuing.
[Termination of CUDA Kernel 0 (mangleGPU) on Device 0]
[Switching to Thread 47173892559264 (LWP 1072)]
```

```
Breakpoint 2, mangleCPU (instr=0xaf2d3e0 "tester", outstr=0xaf8700 " \206\n", len=6, x=1295995981) at
105     for(int i=0; i<len; i++)
```

We're now at Breakpoint 2 (mangleCPU) which means we're now back on a CPU thread and there is no longer an active GPU thread. You can verify this:

```
(cuda-gdb) thread
[Current thread is 2 (Thread 47173892559264 (LWP 1072))]
(cuda-gdb) cuda thread
No CUDA kernel is currently running.
```

We can now step through this function line by line and inspect variables:

```
(cuda-gdb) n
106     MANGLE(instr,outstr,i,len,x);
(cuda-gdb) n
105     for(int i=0; i<len; i++)
(cuda-gdb) n
106     MANGLE(instr,outstr,i,len,x);
(cuda-gdb) n
105     for(int i=0; i<len; i++)
(cuda-gdb) p i
$7 = 1
(cuda-gdb) n
106     MANGLE(instr,outstr,i,len,x);
```

```
(cuda-gdb) n
105     for(int i=0; i<len; i++)
(cuda-gdb) print i
$8 = 2
```

Now that we're done inspecting the code, let's exit the debugger. We can either continue and let the program finish executing or just type quit at the next command prompt:

```
(cuda-gdb) continue
Continuing.
Current date/time: (1295995981) Tue Jan 25 17:53:01 2011
Input string:      tester
CPU result:        teetet
GPU result:        teetet

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

For more info on how to use the CUDA debugger please consult the cuda-gdb user manual.

1.8 Using CUDA Memcheck to Detect Memory-Access Errors

The CUDA memcheck tool helps to detect errors in your code related to memory-access issues. There are two ways to use the CUDA memcheck utility. You can use the cuda-memcheck command line tool to simply run your code and report the errors it finds:

```
cuda-memcheck example -string='hi there class'

===== CUDA-MEMCHECK
Using device 0: Tesla T10 Processor
Current date/time: (1296164555) Thu Jan 27 16:42:35 2011
Input string:      hi there class
CPU result:        sishtiitetst c
GPU result:        sishtiitetst c
===== ERROR SUMMARY: 0 errors
```

This approach will simply print out any errors it found. In this case since the example.cu code included in Homework 0 does not have any memory errors the number of errors reported is 0.

Another option is to use the memcheck utility within the CUDA debugger. This has the advantage that when a memory-access error is detected cuda-gdb will immediately drop you to a shell and allow you to inspect the state of the world. This approach is a bit more complicated given that you have to launch and manage the debugger but it can be extremely powerful when fixing those hard to find bugs.

```
(cuda-gdb) set cuda memcheck on
(cuda-gdb) r
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) p &x
$1 = (@global int *) 0x42c00

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

```

The above output is from the code included in the [cuda-memcheck user manual](#). This code purposefully makes misaligned and out of bound memory accesses in order to demonstrate the behavior within cuda-gdb.

1.9 Profiling CUDA code using the Visual Profiler

The CUDA profiler

- Launch the CUDA visual profiler using the “computeprof” command
- In the dialog that comes up press the “Profile application” button in the “Session” pane.
- In the next dialog that comes up type in the full path to your compiled CUDA program in the “Launch” text area.
- Provide any arguments to your program in the “Arguments” text area. Leave this blank if your code doesn’t take any arguments.
- Make sure the “Enable profiling at application launch” and “CUDA API Trace” settings are checked
- Press the “Launch” button at the bottom of the dialog to begin profiling.

The profiler will now run and analyze your code execution times, memory usage patterns, etc for each function in your code. When it’s finished it will display these stats in an excel spreadsheet-like fashion. You can also plot any given column of data for each method by right-clicking the column and selecting “Column plot”. This is useful in determining which functions are your bottlenecks and might need fixing or refactoring.

For more info on how to use the CUDA debugger please consult the [CUDA visual profiler user manual](#).

1.10 Source Code Control (GIT)

There is a git server setup by SEAS for this course that is available to you while you develop the solutions to the homework problems and for your final projects. Time permitting, let’s take a quick look at using git for simple version control.

- One of the most used distributed version control systems
- Originally written by Linus Torvalds for supporting Linux Kernel development
- Available in most free software distributions
- Available for many operating systems (including Linux/Mac/Windows)
- Forges based on it available: Github, Gitorious, etc.

Links:

- GIT Homepage (<http://git-scm.com/>)

- Mac GIT Client (<http://code.google.com/p/git-osx-installer/>)
- Windows GIT Client (<http://code.google.com/p/msysgit/>)

WELCOME TO CS264 LAB 2

Justin Riley (staff 'at' cs264 'dot' org)
Software Tools for Academics and Researchers
Office of Educational Innovation and Technology
Massachusetts Institute of Technology

2.1 Office Hours

Date/Time: Fridays 3:00pm - 5:00pm

Location: NE-48-308, 77 Massachusetts Ave, Cambridge, MA 02139

2.2 Downloading the Lab Slides

For those that are interested in looking over the slides for the first two labs you can download them [here](#)

2.3 Sign-up for an Amazon Web Services Account

If you haven't already done so please sign-up for an Amazon Web Services (AWS) account and provide your AWS user id in the [homework 0 survey](#). Please see homework 0 for detailed instructions on how to sign up for an Amazon Web Services account and also how to sign up for the Elastic Compute Cloud (EC2) web service.

2.4 Homework 0 Survey

If you haven't already done so please fill out and submit the [homework 0 survey](#).

2.5 Resonance GPU Cluster

Several people have had issues connecting to the resonance cluster:

```
$ gpu-login
Your job 2432796 ("QLOGIN") has been submitted
waiting for interactive job to be scheduled ...
Your interactive job 2432796 has been successfully scheduled.
Establishing /opt/gridengine/bin/rocks-qlogin.sh session to host cuda-5-0.local ...
Connection closed by 10.101.43.244
/opt/gridengine/bin/rocks-qlogin.sh exited with exit code 255
```

This particular error occurs because you haven't yet been given full access to the resonance CUDA cluster. IRCS has been kind enough to relax the requirements for full access and has also removed the need to enable SSH Agent forwarding. This means anyone that is able to login to resonance.seas.harvard.edu should be able to use gpu-login without issues. If you're still having issues logging in to resonance or using the gpu-login command please send an email to ircshelp@seas.harvard.edu.

Another commonly reported error:

```
$ gpu-login
Your job 2433281 ("QLOGIN") has been submitted
waiting for interactive job to be scheduled ...timeout (5 s) expired while waiting on socket fd 4
```

This issue is generally caused by having another already active session running. This can happen if you don't exit out of a previous gpu-login session correctly. To prevent this make sure that you always type "exit" when you're done using a gpu-login session. If this error occurs you need to first cancel the old session:

```
$ qstat
job-ID prior name user state submit/start at queue slot
-----
2433880 0.56895 QLOGIN jtriley r 02/03/2011 15:40:41 gpu-interactive.q@cuda-1-1.loc

$ qdel 2433880
```

After you've cancelled your previous gpu-login session you should now be able to use the gpu-login command without error assuming there are enough available GPU nodes to schedule your gpu-login session.

2.6 Using Graphical Applications on the Resonance GPU Cluster

In order to use graphical applications such as the CUDA Profiler and the DDD debugger you will need to enable X11 forwarding in your SSH client configuration.

On Linux/Mac this can be accomplished using the -X flag:

```
$ ssh -i ~/.ssh/myprivkey -A -X myuser@resonance.seas.harvard.edu
```

If you wish to set this permanently you can do this using the ~/.ssh/config file. If the 'config' file does not exist in the ~/.ssh directory you will need to create it. Then you can permanently enable the -X option anytime you ssh to resonance by putting the following in ~/.ssh/config:

NOTE: On Mac you will need to install the X server that comes with Mac. If you do not already have X installed you can install it from the Mac OSX installer CD.

```
$ vim ~/.ssh/config
(add the following lines to the file)
Host resonance*
    HostName resonance.seas.harvard.edu
    IdentityFile /path/to/your/privkey
    ForwardX11 yes
    ForwardAgent yes
```


On Windows you will need to download and install an X server. [Xming](#) is freely available and works well with [Putty](#). After you've installed and started the X server you will need to configure Putty to "Enable X11 forwarding".

[Here's a guide on how to configure Xming and Putty on Windows](#) (thanks to Robert Bowden for posting the link on the forums!)

2.7 Debugging CUDA using cuda-ddd on Resonance

In this Lab we will take a look at how to remotely use the Data Display Debugger (DDD) application to graphically debug our CUDA code. DDD allows you to debug and monitor variables via a graphical console. You will still need to know how to switch between CUDA threads as we did in the first Lab using `cuda-gdb`, however, printing and monitoring variables should be much easier with DDD.

The first step is to login to resonance with X11 forwarding enabled and run the 'gpu-login' command:

```
$ ssh -i ~/.ssh/myprivkey -X -A username@resonance.seas.harvard.edu
$ gpu-login
```

Once you've successfully been logged in to a GPU-enabled machine you will then need to add the `cuda-ddd` package in order to access the 'cuda-ddd' command:

```
$ module load packages/ddd/3.3.12
```

If you configured your `~/.bashrc` or `~/.bash_profile` to permanently load packages as discussed in HW0 you should be able to permanently load `cuda-ddd` using:

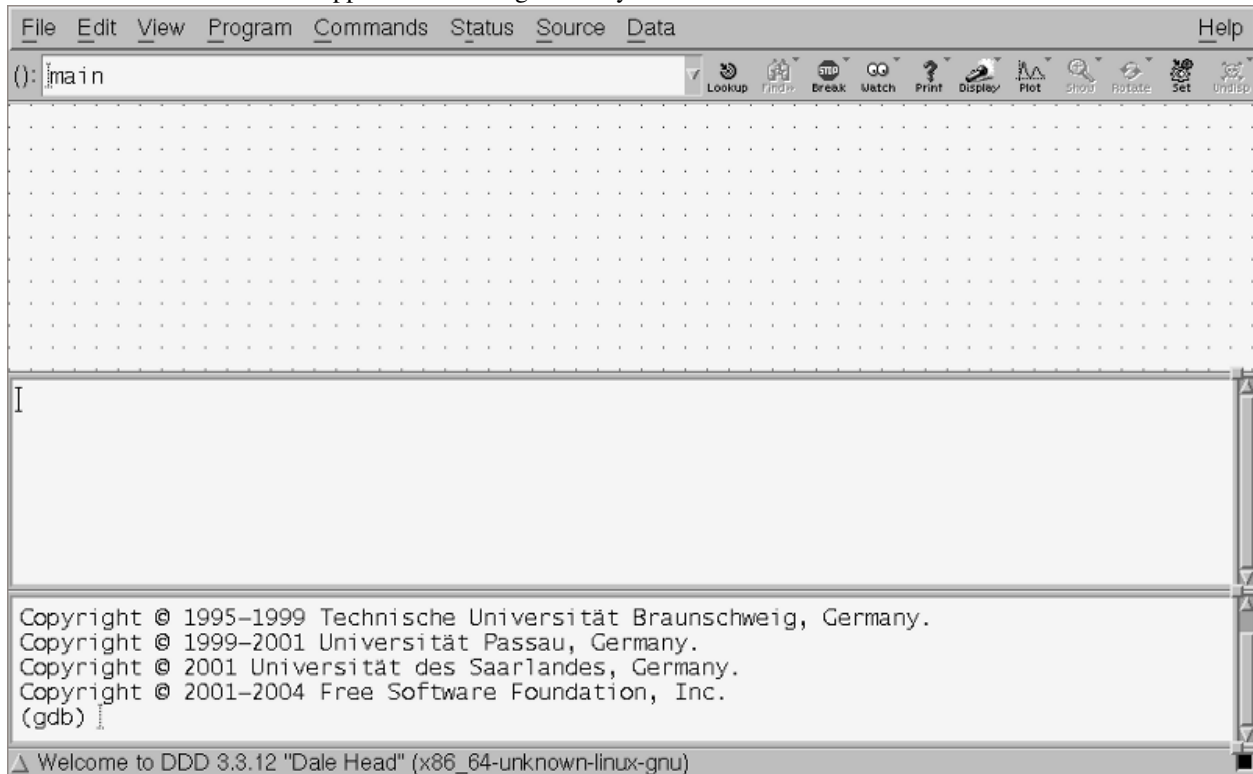
```
$ module initadd packages/ddd/3.3.12
```

Once you've added the `ddd` package you should now be able to run the `cuda-ddd` command:

```
$ cuda-ddd
```

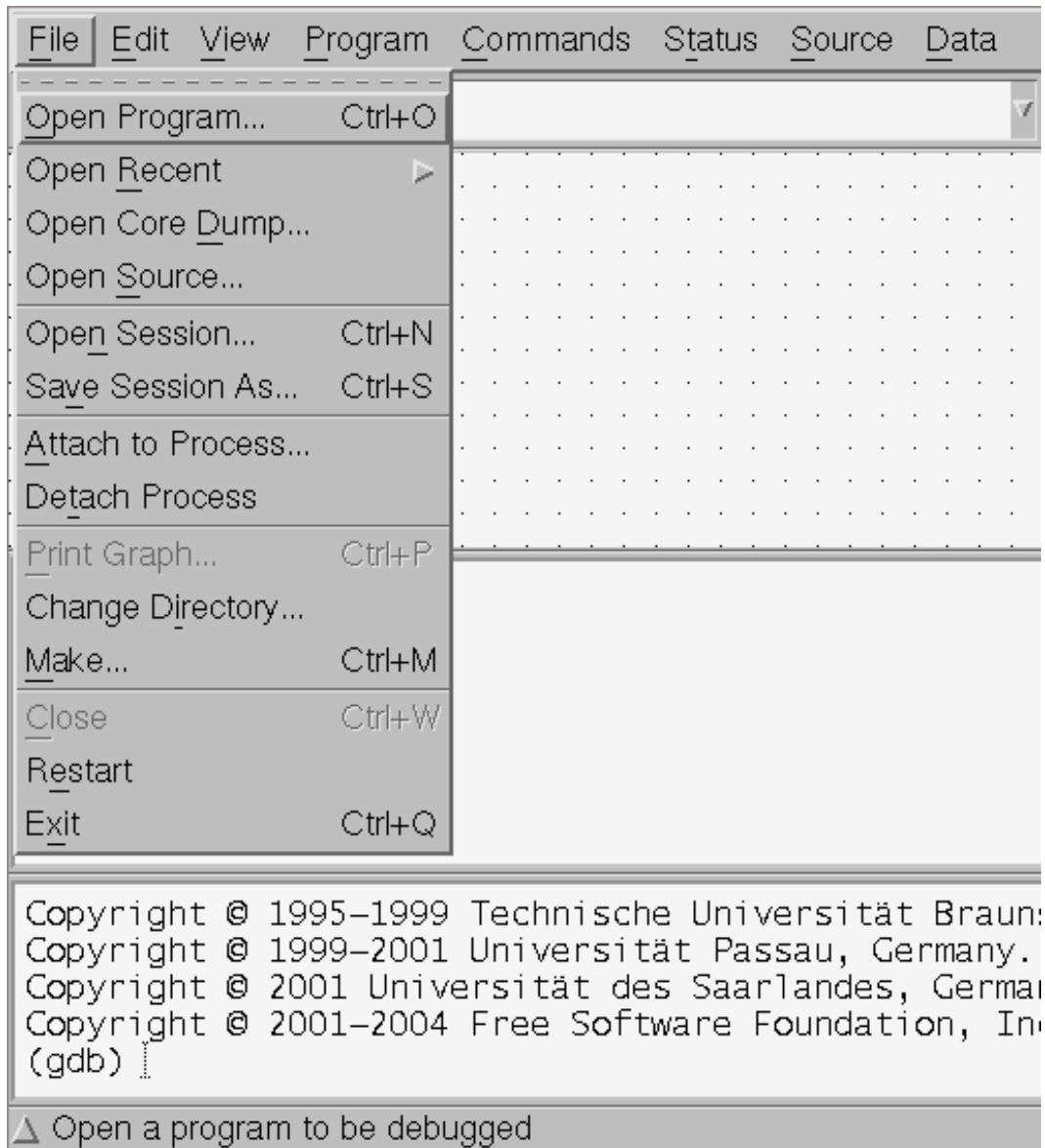
Provided you've successfully configured X11 forwarding you should now see a window popping up on your desktop.

This window is the `cuda-ddd` application running remotely on resonance:

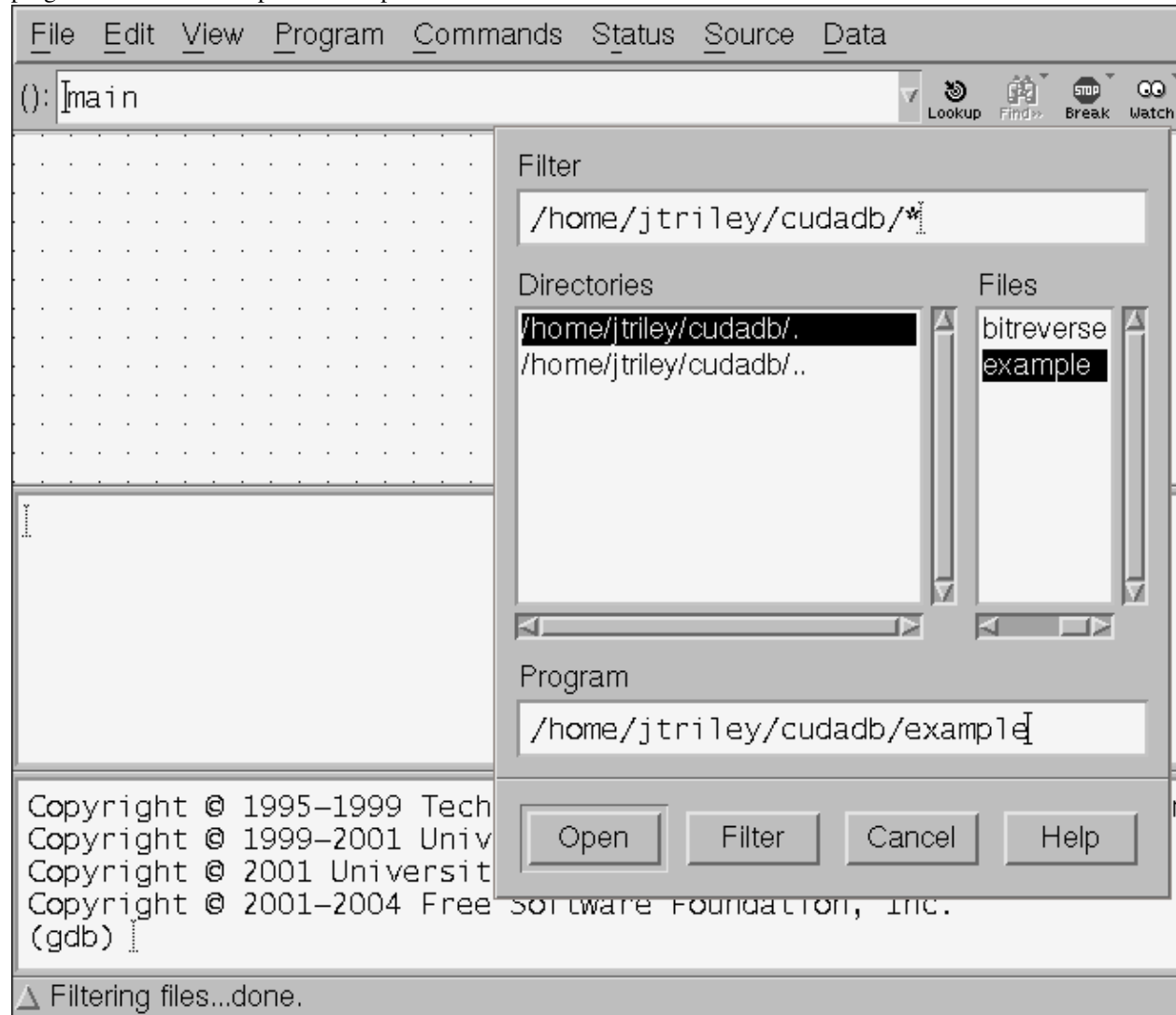


Now that we've launched ddd let's open the example program from HW0 (**don't forget to compile with -g -G flags!**).

To do this go to File->Open Program:



This will prompt you with a dialog asking you to select the program you want to run. Select the example CUDA program from HW0 and press the “Open” button:



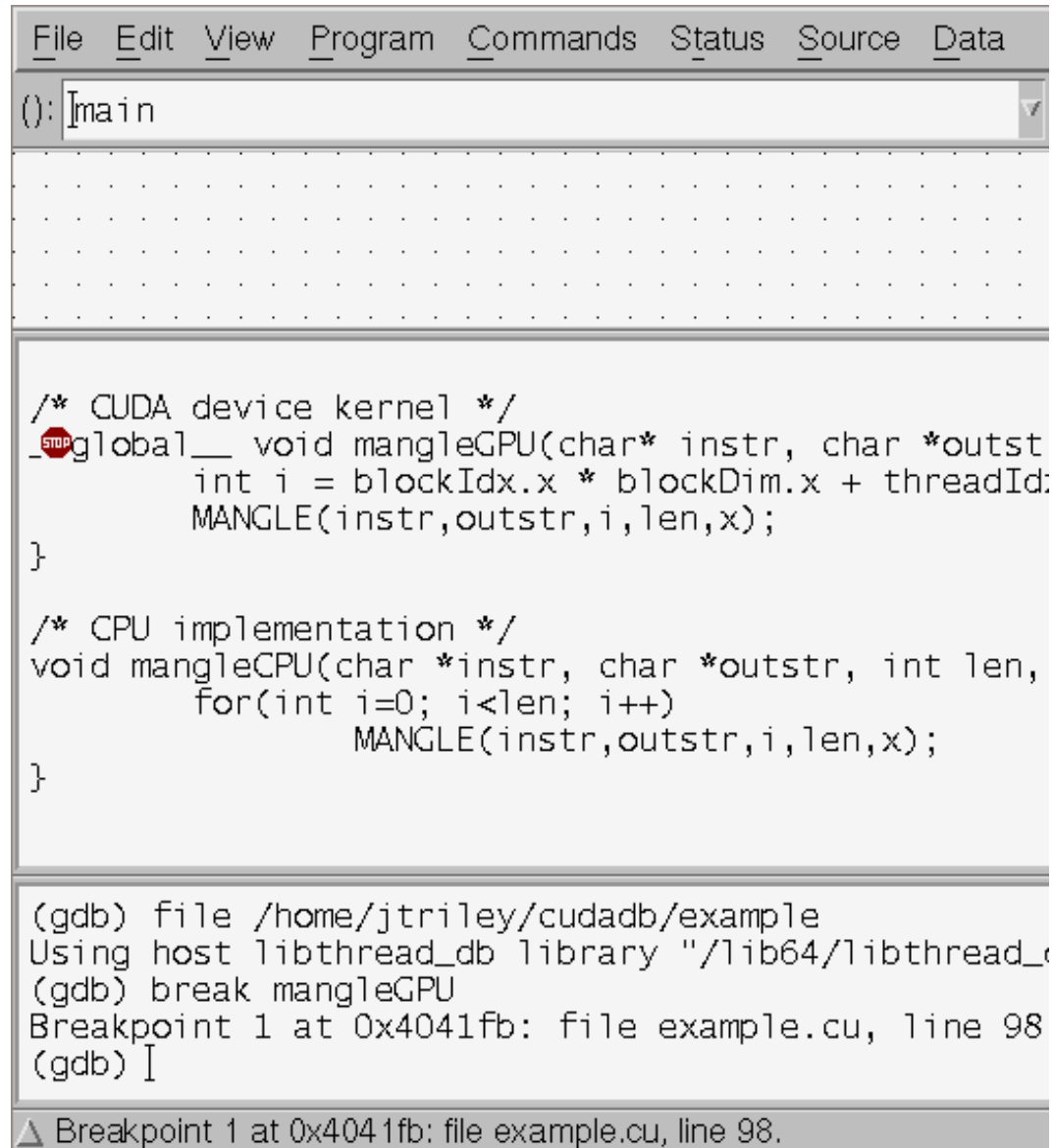
Once you've opened the example program you should see:

The screenshot shows the cuda-ddd debugger window. The top menu bar includes File, Edit, View, Program, Commands, Status, Source, Data, and Help. Below the menu is a toolbar with icons for various debugging actions. The main window displays the source code of example.cu, which contains two functions: mangleGPU (a CUDA device kernel) and mangleCPU (a CPU implementation). The bottom panel shows the GDB command window with the following text:

```
Copyright © 2001 Universität des Saarlandes, Germany.
Copyright © 2001-2004 Free Software Foundation, Inc.
(gdb) file /home/jtriley/cudadb/example
Using host libthread_db library "/lib64/libthread_db.so.1".
(gdb) |
```

At the very bottom of the window, a status bar indicates: `△ Disassembling location 0x4053fa to 0x4054fa...done.`

The next step is to create a breakpoint. You can do this by typing a ‘break’ command in the cuda-gdb console in the bottom pane of DDD:



The screenshot shows the DDD interface with the following components:

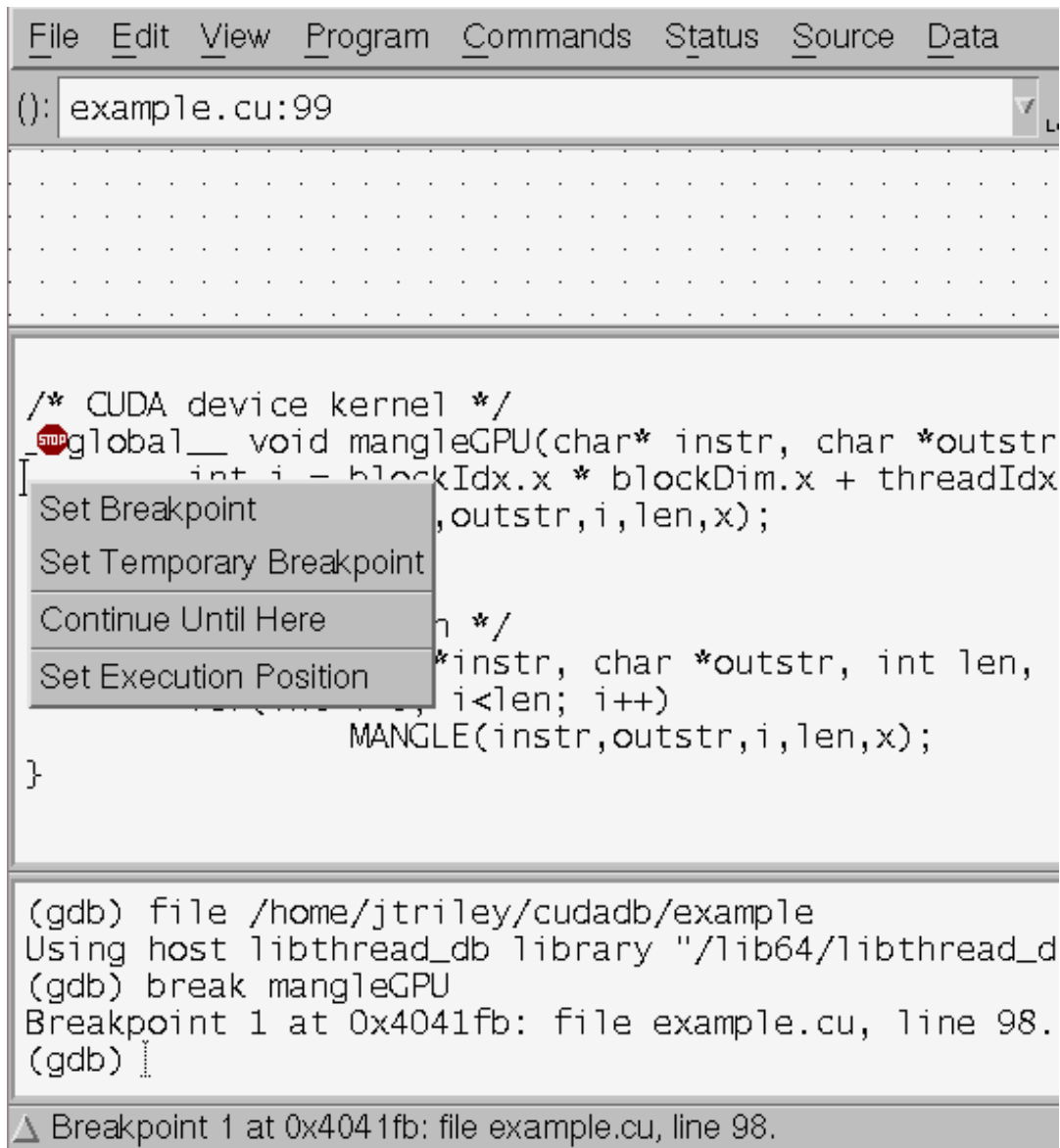
- Menu Bar:** File, Edit, View, Program, Commands, Status, Source, Data.
- Symbol Table:** A dropdown menu showing '(): main'.
- Source Code Window:** Displays the source code of 'example.cu'. The code includes a CUDA device kernel and a CPU implementation. A red 'STOP' icon is visible next to the start of the GPU function.
- Console Window:** Shows the gdb commands and their output:

```
(gdb) file /home/jtriley/cudadb/example
Using host libthread_db library "/lib64/libthread_db.so.1"
(gdb) break mangleGPU
Breakpoint 1 at 0x4041fb: file example.cu, line 98
(gdb) I
```
- Status Bar:** Displays 'Breakpoint 1 at 0x4041fb: file example.cu, line 98.'

```
/* CUDA device kernel */
global__ void mangleGPU(char* instr, char *outstr,
                        int i = blockIdx.x * blockDim.x + threadIdx.x) {
    MANGLE(instr, outstr, i, len, x);
}

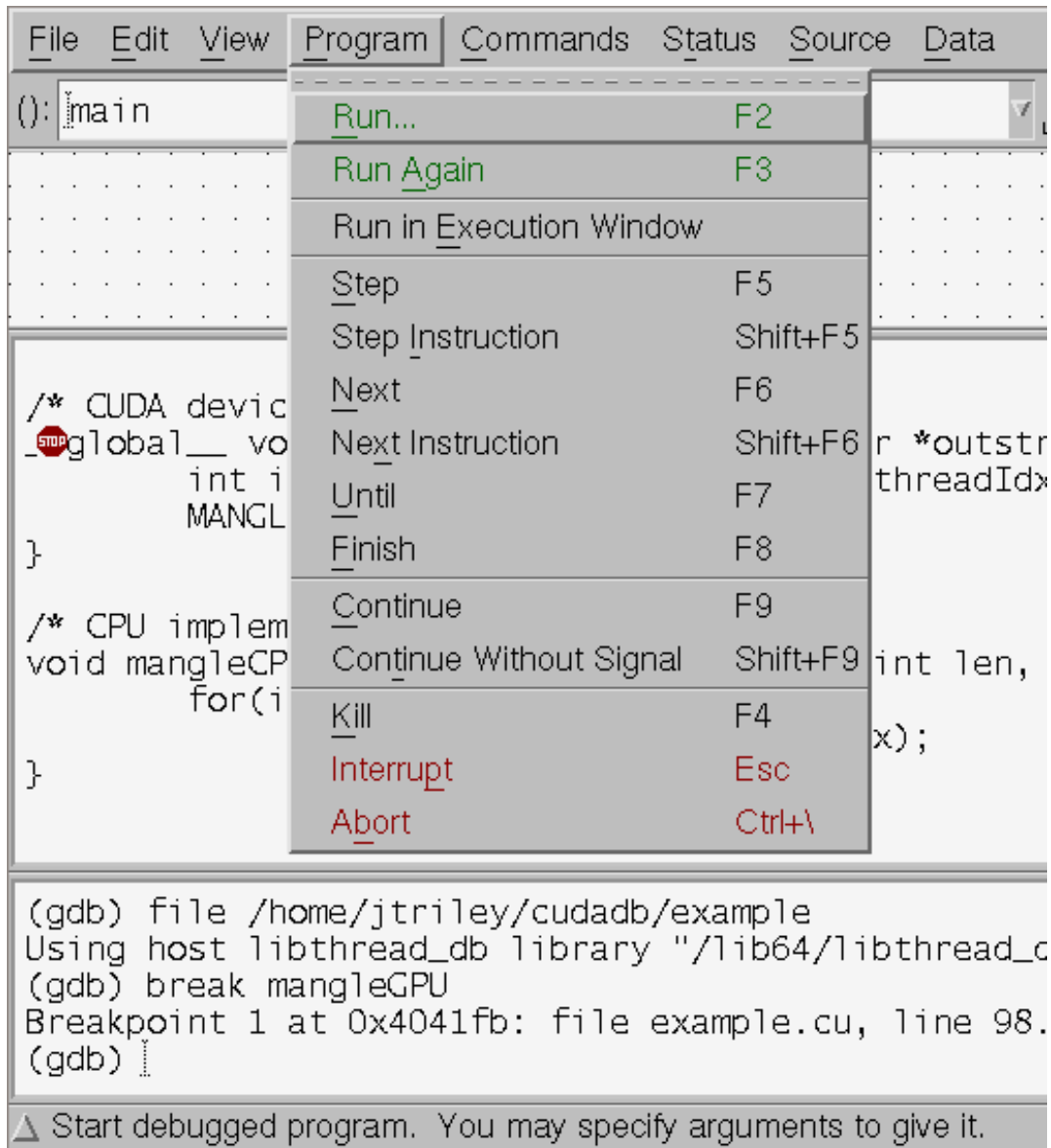
/* CPU implementation */
void mangleCPU(char *instr, char *outstr, int len,
               for(int i=0; i<len; i++)
                   MANGLE(instr, outstr, i, len, x);
}
```

You can also set a breakpoint graphically by right clicking the line you're interested in (make sure to right-click in the whitespace before the line) and selecting "Set breakpoint":

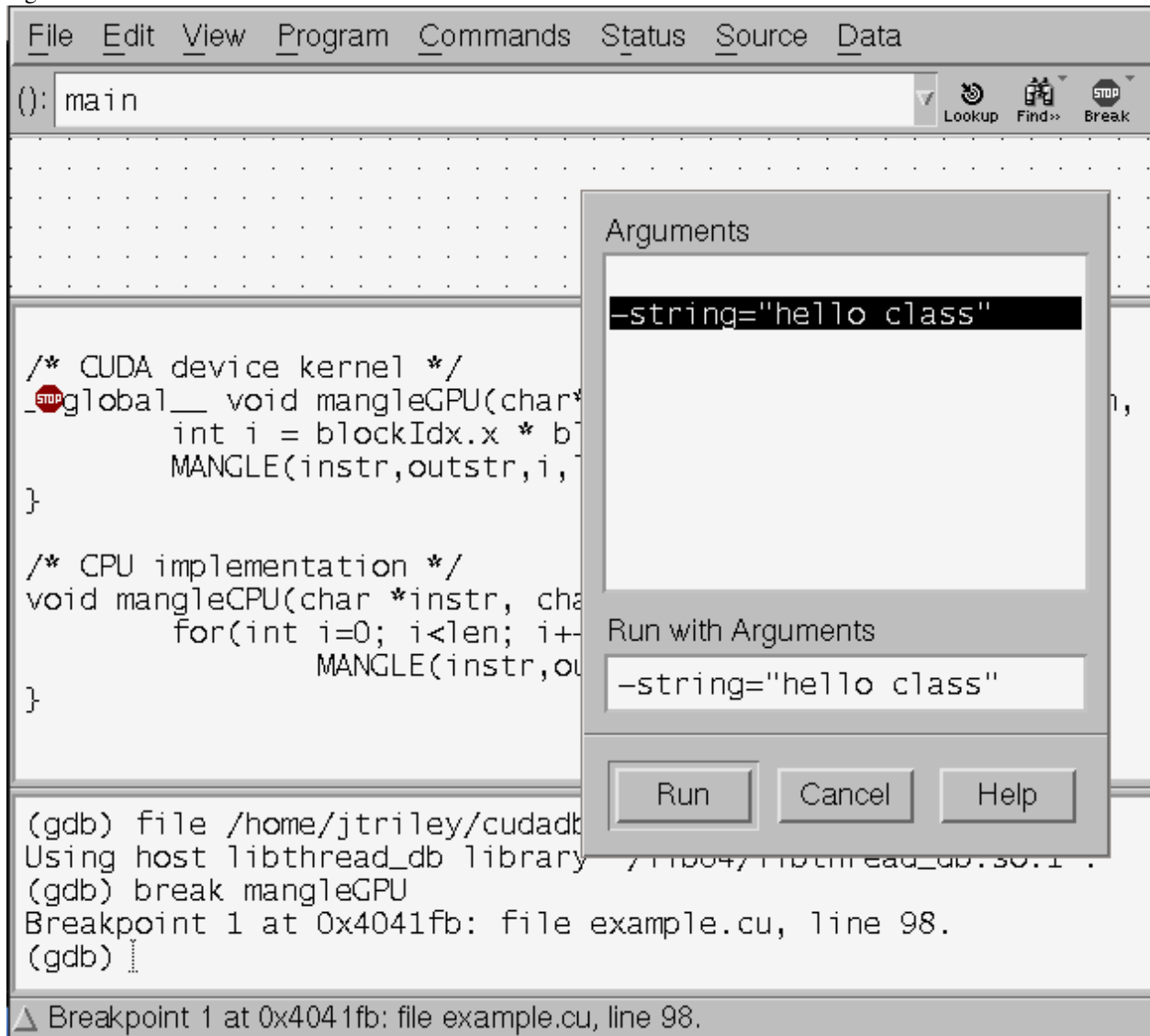


In this example we've added a breakpoint at the *mangleGPU* function.

Now that we have a breakpoint set it's time to run your program. To do this go to Program->Run...:



This will prompt you with a dialog to enter arguments. If your code does not take arguments you may simply press the “Run” button to start your program within the CUDA debugger. The example code in HW0 requires a -string argument so we add it here:



After you press the “Run” button in the arguments dialog your program will run and the debugger will stop at the first defined breakpoint (mangleGPU in this case). You can now use your mouse to hover over variables and inspect their current values:

The screenshot shows a debugger window with the following components:

- Menu Bar:** File, Edit, View, Program, Commands, Status, Source, Data.
- Address Bar:** Shows the current location as `() : example.cu:102`.
- Source View:** Displays the source code of `example.cu`. A red stop icon and a green arrow indicate a breakpoint is set at line 102. The code is as follows:


```

/* CUDA device kernel */
__global__ void mangleGPU(char* instr, char *outstr,
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    MANGLE(instr, outstr, {x = 0, y = 0});
}

/* CPU implementation */
void mangleCPU(char *instr, char *outstr, int len,
    for(int i=0; i<len; i++)
        MANGLE(instr, outstr, i, len, x);
}
      
```
- Breakpoint View:** Shows the details of the breakpoint:

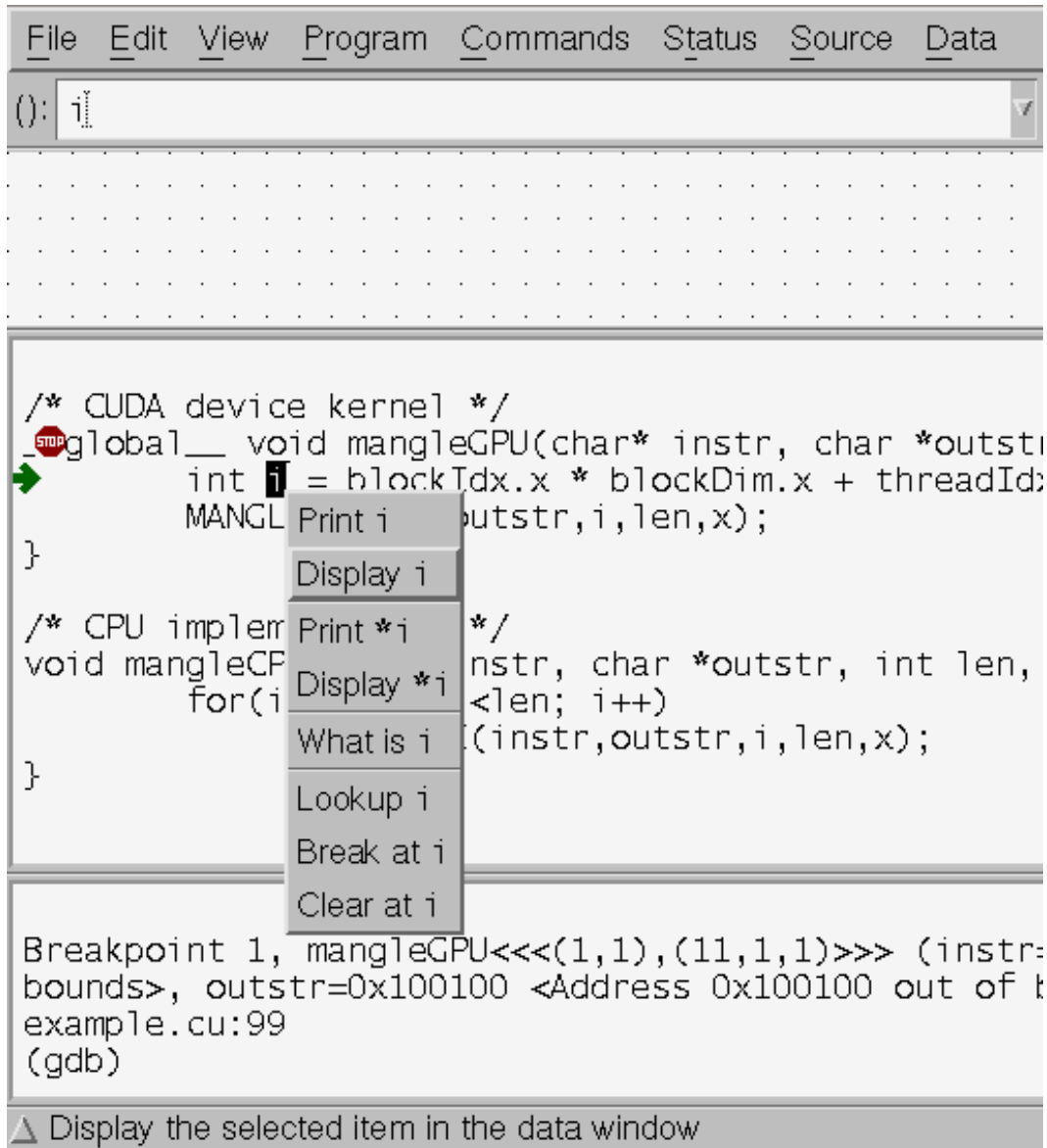

```

Breakpoint 1, mangleGPU<<<(1,1),(11,1,1)>>> (instr:
bounds>, outstr=0x100100 <Address 0x100100 out of b
example.cu:99
(gdb) I
      
```
- Variable View:** At the bottom, it shows the current state of the `blockIdx` variable:

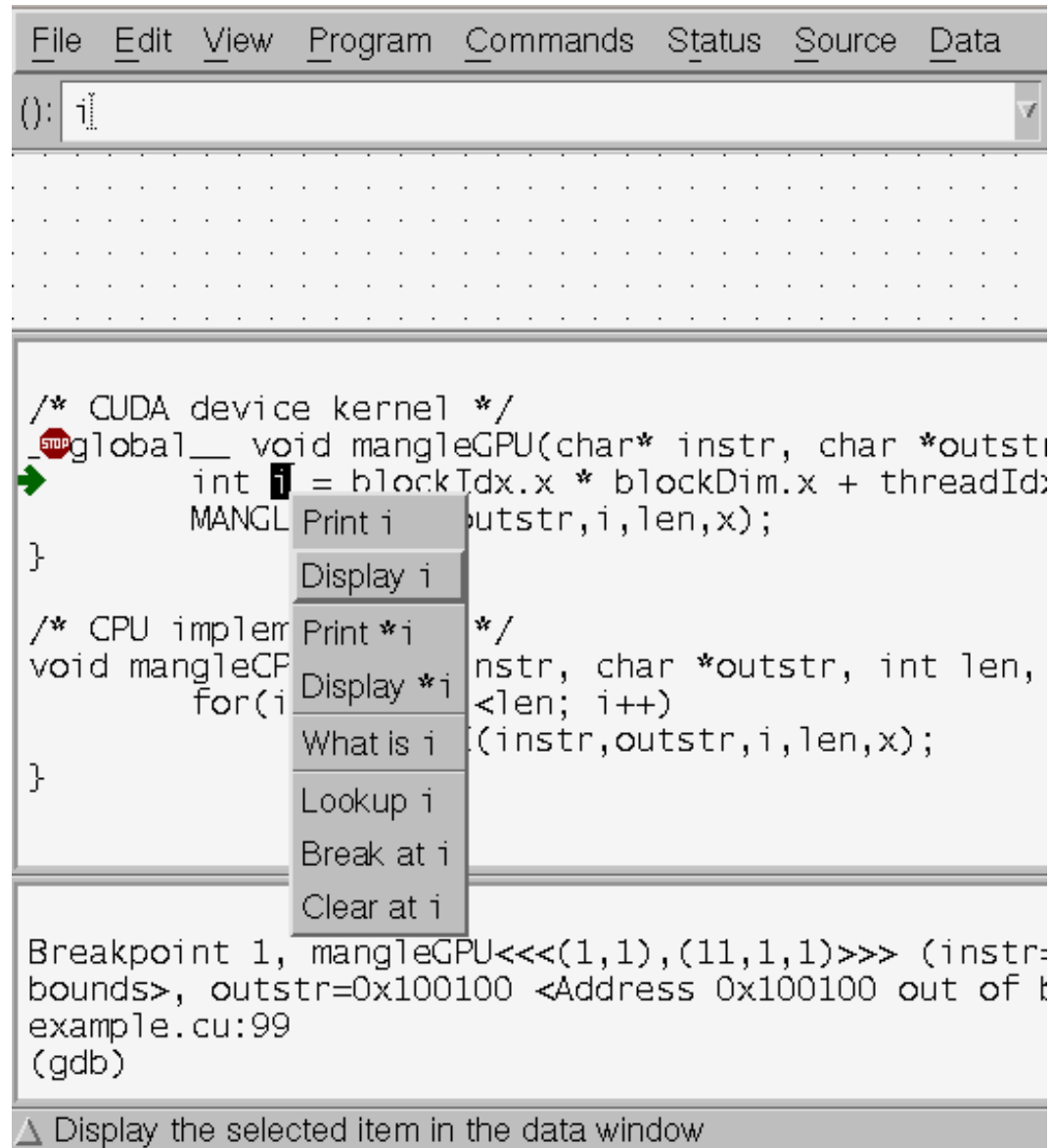

```

Δ blockIdx = {x = 0, y = 0}
      
```

In addition to highlighting we can monitor variables using DDD. This will allow us to select many different variables we're interested in and inspect them all at once each time we step through the code or switch GPU threads. In order to add a monitor for a variable simply highlight the variable you're interested in, right-click, and select *Display* *<varname>*:



Notice how a new variable gets added to the top window pane in DDD. This pane is the variable monitoring pane. Each time you right click a variable and select the *Display <varname>* item it should show up in the variable monitoring pane.



Now that we have our monitors set up, let's switch GPU threads and see how the variable monitor pane updates all the values we're interested in.

The screenshot shows the cuda-ddd debugger interface. At the top is a menu bar with File, Edit, View, Program, Commands, Status, Source, and Data. Below the menu is a search bar containing '(): threadIdx'. The main area is divided into three sections. The top section contains four variable monitors: 1: i (value 2), 2: blockIdx (x=0, y=0), 3: blockDim (x=11, y=1, z=1), and 4: threadIdx (x=2, y=0, z=0). The bottom section shows the source code for the mangleGPU kernel, with a green arrow pointing to the MANGLE function call. The bottom-most section shows the gdb command line with the following commands: (gdb) n, (gdb) cuda thread 2, [Switching to CUDA Kernel 0 (device 0, sm 0, warp (2,0,0))], and (gdb) I. The status bar at the bottom indicates 'Updating displays...done.'

```
File Edit View Program Commands Status Source Data
(): threadIdx

1: i
2

2: blockIdx
x = 0
y = 0

3: blockDim
x = 11
y = 1
z = 1

4: threadIdx
x = 2
y = 0
z = 0

/* CUDA device kernel */
__global__ void mangleGPU(char* instr, char *outstr,
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    MANGLE(instr, outstr, i, len, x);
}

/* CPU implementation */
void mangleCPU(char *instr, char *outstr, int len,
    for(int i=0; i<len; i++)

(gdb) n
(gdb) cuda thread 2
[Switching to CUDA Kernel 0 (device 0, sm 0, warp (2,0,0))]
(gdb) I

Updating displays...done.
```

This concludes our introduction to using the DDD program to debug CUDA code. For more information on all of DDD's features please have a look at the [DDD user manual](#).

2.8 Using CUDA Memcheck to Detect Memory-Access Errors

The CUDA memcheck tool helps to detect errors in your code related to memory-access issues. There are two ways to use the CUDA memcheck utility. You can use the cuda-memcheck command line tool to simply run your code and report the errors it finds:

```
cuda-memcheck example -string='hi there class'

===== CUDA-MEMCHECK
Using device 0: Tesla T10 Processor
Current date/time: (1296164555) Thu Jan 27 16:42:35 2011
Input string:      hi there class
CPU result:        sishtiitetst c
GPU result:        sishtiitetst c
===== ERROR SUMMARY: 0 errors
```

This approach will simply print out any errors it found. In this case since the example.cu code included in Homework 0 does not have any memory errors the number of errors reported is 0.

Another option is to use the memcheck utility within the CUDA debugger. This has the advantage that when a memory-access error is detected cuda-gdb will immediately drop you to a shell and allow you to inspect the state of the world. This approach is a bit more complicated given that you have to launch and manage the debugger but it can be extremely powerful when fixing those hard to find bugs.

```
(cuda-gdb) set cuda memcheck on
(cuda-gdb) r
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) p &x
$1 = (@global int *) 0x42c00

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

The above output is from the code included in the [cuda-memcheck user manual](#). This code purposefully makes misaligned and out of bound memory accesses in order to demonstrate the behavior within cuda-gdb.

2.9 Profiling CUDA code using the Visual Profiler

We will learn about profiling CUDA code in depth during upcoming lectures. For now we will take a practical look at the CUDA Visual Profiler. Specifically we'll learn how to launch the CUDA Visual Profiler, run our CUDA code, and look at the profiling results.

If you wish to use the CUDA Visual Profiler on the resonance GPU cluster you will need to enable X11 forwarding and then use the `gpu-login` command to access a gpu-enabled node:

```
$ ssh -i ~/.ssh/myprivkey -A -X user@resonance.seas.harvard.edu
$ gpu-login
```

After you've logged in you can launch the CUDA visual profiler using the “`computeprof`” command:

```
$ computeprof
```

- 1 Once you've launched the profiler press the "Profile application" button in the "Session" pane.
- 2 In the next dialog that comes up type in the full path to your compiled CUDA program in the "Launch" field.
- 3 Provide any arguments to your program in the "Arguments" text field. Leave this blank if your code doesn't take any arguments.
- 4 Make sure the "Enable profiling at application launch" and "CUDA API Trace" settings are checked.
- 5 Press the "Launch" button at the bottom of the dialog to begin profiling.

The profiler will now run and analyze your code execution times, memory usage patterns, etc for each function in your code. When it's finished it will display these stats in an excel spreadsheet-like fashion. You can also plot any given column of data for each method by right-clicking the column and selecting “Column plot”. This is useful in determining which functions are your bottlenecks and might need fixing or refactoring.

For more info on how to use the CUDA debugger please consult the [CUDA visual profiler user manual](#).

2.10 Source Code Control Using GIT

What is GIT?

- One of the most used distributed version control systems
- Originally written by Linus Torvalds for supporting Linux Kernel development
- Available in most free software distributions
- Available for many operating systems (including Linux/Mac/Windows)
- Forges based on it available: Github, Gitorious, etc.

Where does the name come from?

From wikipedia: Linus Torvalds has quipped about the name “git”, which is British English slang for a stupid or unpleasant person: “I’m an egotistical <expletive deleted>, and I name all my projects after myself. First Linux, now git.”

This section gives a very quick introduction to using git to manage local repositories. We will cover using git with remote servers later on.

It's actually very easy to version control *any* directory on your computer using git. For the purpose of this walk-through let's make a blank project directory and enable version control via GIT:

```
$ mkdir mynewproj
$ cd mynewproj
$ git init
Initialized empty Git repository in /home/myuser/mynewproj/.git/
```

Now that we’ve initialized version control in the *mynewproj* directory, let’s add some project files:

```
$ vim ninjacuda.cu
(write some CUDA code and save)
```

Now that we’ve worked so hard at creating a project file let’s add it to version control:

```
$ git add ninjacuda.cu
```

The *ninjacuda.cu* file is now in a “staging area” waiting to be committed. You can see this by running the “git status” command:

```
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   ninjacuda.cu
#
```

Now that the *ninjacuda.cu* has been added to the staging area we can make changes to this file and check what’s changed since we last ran “git add” on the file using the “git diff” command.

```
$ echo "whoops didnt mean to mess up the file" >> ninjacuda.cu
$ git diff
diff --git a/ninjacuda.cu b/ninjacuda.cu
index 37d4e6c..a4b7aa5 100644
--- a/ninjacuda.cu
+++ b/ninjacuda.cu
@@ -1,2 @@
  hi there
+whoops didnt mean to mess up the file
```

We can also use the “git status” command to check what files have been updated. Here we see that we’ve indeed modified the *ninjacuda.cu* code:

```
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   ninjacuda.cu
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   ninjacuda.cu
#
```

In this example we didn’t mean to add this new line to our GPU code. Fortunately we can always revert back to the state it was in when we last ran “git add” on the file using the “git checkout” command:


```
$ git checkout ninjacuda.cu
```

This is useful when you’ve made changes to a file that you don’t wish to keep and just want to get back to the ‘working version’:

If we had decided to *keep* the changes we made we would simply have run the “git add” command again and it would update the “staging area” with the latest changes to the file.

Now that we’re happy with the *ninjacuda.cu* file and have used “git add” to update the “staging area” with the latest changes let’s go ahead and commit it to version control so that we can track our progress and always revert back to this working state if we need to in the future:

Running “git status” again we should see the new file staged to be committed:

```
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   ninjacuda.cu
#
```

Running “git commit” will launch an editor (determined by the \$EDITOR environment variable) allowing us to type in a useful log message to attach to the commit. You will also see the (commented) output of the “git status” command below the first line in the editor:

```
$ git commit
my first commit
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   ninjacuda.cu
#
[master (root-commit) e4bc1ba] my first commit
1 files changed, 1 insertions(+), 0 deletions(-)
create mode 100644 ninjacuda.cu
```

Type a brief message explaining the changes you made and exit the editor. Congrats, you just made your first commit!

If we look at the logs we can now take a look at the history of our (very small) project:

```
commit e4bc1baa38110c72409d9b5dcc3005d80b4c6593
Author: John Smith <john.s@example.com>
Date:   Thu Feb 3 18:33:55 2011 -0500

    my first commit
```

We’ll learn later on how to use “remote” git repositories so that you can “push” and “pull” your commits to and from a remote git server. Having a remote git server allows you to back-up your work off-site and also allows you to access your work from anywhere.

NOTE: There is a git server setup by SEAS for this course that is available to you while you develop the solutions to the homework problems and for your final projects.

Useful GIT Links:

- GIT Homepage (<http://git-scm.com/>)
- Mac GIT Client (<http://code.google.com/p/git-osx-installer/>)
- Windows GIT Client (<http://code.google.com/p/msysgit/>)

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*