GKLEE User Manual

Gauss Research Group,
School of Computing,
University of Utah,
Salt Lake City, UT 84112

March 9, 2012

Contents

1 INTRODUCTION

GKLEE[1]is a *concolic* verifier-*cum*-analyzer for CUDA (Compute Unified Device Architecture [1]) C/C++ programs that are typically executed on GPUs (Graphical Processing Unit). GKLEE actually analyzes whole programs, which means that it verifies the "`main`" (CPU) program together with the collection of GPU kernel functions that it calls (we call these programs "GPU programs"). The term "concolic" is a portmanteau of the terms 'Concrete' and 'Symbolic.' Concolic verifiers are gaining widespread adoption in many areas of software testing [2]. They aim to achieve the union of the good features of concrete and symbolic verification methods, while also aiming for the intersection of their disadvantages. GKLEE extends KLEE [3] in many ways. While KLEE provides the basic capabilities for sequential program analysis, GKLEE ("GPU + KLEE") extends KLEE to provide self-contained and powerful facilities for analyzing GPU programs.

1.1 More GKLEE features

GKLEE is a tool that executes the programmer's CUDA application in a symbolic environment. With the user declared symbolic assignment of program variables, GKLEE can execute through all different branches in the code where the predicates are based on the symbolic variables. These executions are output as test cases, with concrete values substituted for the symbolic ones. GKLEE fully executes the CUDA program, including the kernel portion, closely following CUDA semantics and using a canonical schedule for warp execution (which is proven to be sound for race detection).

Using GKLEE's Emacs interface, the user may navigate through these test cases, stepping through both the original source code and the generated LLVM, and gain insight into the behavior of their CUDA program.

For more information on GKLEE, please refer to `http://www.cs.utah.edu/fv/GKLEE/`, which includes usage instructions, examples, a liveDVD with a fully configured environment, a detailed tech report, and our *PPoPP 2012* paper.

The GKLEE Emacs interface works by encapsulating the execution of GKLEE and the LLVM compilers. With a few keystrokes, the user's CUDA program can be compiled into bytecode (at the specified optimization level)

[1] "GKLEE" is pronounced as "G" followed by how "KLEE" is pronounced ("Clay?")

and fed into GKLEE. Feedback is immediately generated on the status of the execution. Once the run is complete, the output is analyzed and the user is presented with a buffer giving a synopsis of the GKLEE run. Each generated trace is listed with statistics on its execution, including a measurement of performance anomalies and indication of any errors. Gklee-mode then allows the user to drill down to the location of uncovered issues. From there, the user may step through the execution, focusing only on the locations, blocks, threads, warps of interest.

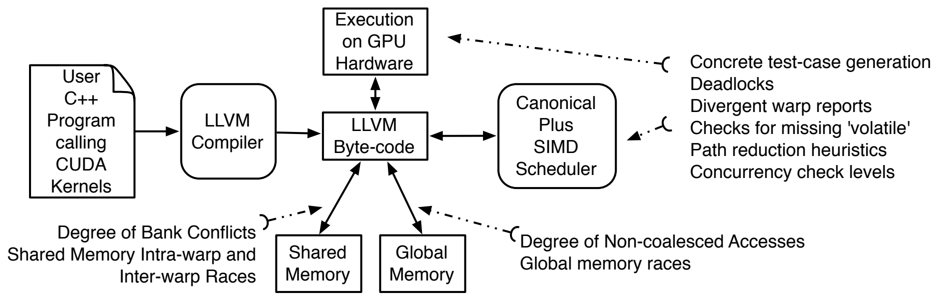

Execution on GPU Hardware
User C++ Program calling CUDA Kernels
LLVM Compiler
LLVM Byte-code
Canonical Plus SIMD Scheduler
Concrete test-case generation
Deadlocks
Divergent warp reports
Checks for missing 'volatile'
Path reduction heuristics
Concurrency check levels
Degree of Bank Conflicts
Shared Memory Intra-warp and Inter-warp Races
Shared Memory
Global Memory
Degree of Non-coalesced Accesses
Global memory races

Figure 1: GKLEE's overall flow and verification features supported

Figure 1 summarizes the overall flow of GKLEE and its verification features. The rest of this manual will detail how these features can be exercised through either an `Emacs` based interface or a command-line based interface.

GKLEE's main features are the following:

- Compiles the program into LLVM
- GKLEE detects and reports occurrences of divergent thread warps (branches inside SIMD paths); these can degrade performance.
- GKLEE finds deadlocks caused by non textually-aligned barrier (`__syncthreads()`) calls.
- GKLEE's symbolic virtual machine can systematically generate concrete test inputs based on path constraint solving.
- GKLEE employs GPU-specific heuristics for reducing the number of tests generated. It also retain's KLEE's flag options, which includes the ability to either retain or eliminate redundant test inputs that drive execution through the same path.
- A feature of GKLEE's flow is the ability to process these tests to a form that can be run on the GPU hardware. (We will send you these scripts if you are interested.)

- We target two classes of memory access inefficiencies, namely non-coalesced global memory accesses and shared memory accesses that result in bank conflicts, and show how GKLEE can spot these inefficiencies, also "understanding" platform rules (i.e., compute capability 1.x or 2.x).
- GKLEE's VM incorporates the CUDA memory model within its concolic execution framework, while accurately modeling the SIMD concurrency of GPUs. It can scale to large concurrent executions involving 2K threads mainly because it uses a new scheduling method called *canonical plus SIMD*. This scheduling method is sound, while generating only *one* schedule (as opposed to an exponential number of schedules in traditional approaches).
- GKLEE handles many C++/CUDA features including: struct, class, template, pointer, inheritance, CUDA's variable and function derivatives, and CUDA specific functions.
- GKLEE's analysis occurs on LLVM byte-codes (also targeted by Fortran and Clang). Byte-code level analysis can help cover pertinent compiler-induced bugs in addition to supporting future work on other binary formats.

GKLEE's Emacs mode has several noteworthy features:

- It helps compile user programs to LLVM byte-code and helps launch concolic analysis
- It helps check for races, assertion violations, global memory coalesced accesses, bank conflicting shared memory accesses and warp divergence
- It helps distill the GKLEE output, giving concise program statistics
- It makes all generated execution traces available for exploration
- It provides a stepping mode
- It has filter functionality to only view the threads/blocks/warps/code locations of interest

2 Case Study: Radix Sort

The best way to obtain a comprehensive picture of GKLEE is to dive into an example that came with CUDA SDK 2.0, namely the `radixSort.C` example. We now walk you through all the steps in debugging this example using GKLEE.

- Choose the example radixSort.C in directory

/home/ganesh/gkllee/Gklee/CUDA/Benchmarks/Table-1/SDK2.0/radixSort/radixSort.C

- Now type `ESC-g r`, which means “Run Gklee.”
- An explanation of some of GKLEE’s Emacs-mode buffers:
  - Buffer `*gklee-run-debug*` contains the details of the GKLEE analysis
  - Buffer `*gklee-compile-debug*` contains the details of GKLEE’s compilation
  - Buffer `*gklee-run*` shows red-highlighted occurrences of Write write races within a warp
- We notice from buffer `*gklee-run*` that there are data races involving threads 0 and 4. A single trace file `*test000001.trace*` is listed in this buffer (GKLEE may, in general, produce many tests that cover various control branches). Click just under the red-highlighted error message in this buffer. This pulls up the trace file (source + LLVM) into the `*test000001.trace*` buffer.
- Notice that the bottom two windows (buffers) have changed into the following: `*gklee-available-filters*` and `*gklee-active-filters*`.
- Suppose you wish to hide the LLVM-level instructions. Focus (“click”) on buffer `*test000001.trace*`, and type `ESC-gta` (toggle away the LLVM assembly). The LLVM instructions should hide away. Try this again, and the LLVM instructions re-appear.
- Now let us proceed to identify the bug. Generally a user is interested in quickly discovering where the data race happens by watching the execution around the site of the bug. We already know that Thread 0 and Thread 4 are involved in the race. So how about filtering away the other threads? Here is how you accomplish the filtering:
  - Set focus on `*gklee-available-filters*` (by clicking anywhere in this buffer).
  - Search (using Emacs search commands) for ”Thread Filters”. Click on ”All_Threads”. This makes all threads part of the `*gklee-active-filters*` buffer. This also makes all the trace entries to disappear!
  - Now add back threads T0 and T4 by going to the buffer `*gklee-active-filters*` and selecting T0 and T4 (click on them). You’ll find that these two entries move to the `*gklee-available-filters*` buffer.
  - At the same time, you also see the `*test000001.trace*` trace buffer come alive, now containing only the activities of these two

threads.

- Click on one of the red lines which reads Line 499, Block 0, Thread 4, File radixsort.C
- You will now see that the cursor is set on radixsort.C at line 499
- Now paint a region of text ending at line 499, going up a few lines (as far behind as you want)
- Right-click (on a MacBook Pro, two fingers on the pad softly at the top, and a third finger wandering down and clicking towards the right of the pad)
- Presto! The remaining lines go away!
- Now set focus on the `*test000001.trace*` buffer, and type ESC-gss (enter single-stepping mode)
- Single-step by typing "n" or `ESC-gne` (after setting focus on the trace buffer `*test000001.trace*`).
- You can now see how the error is arrived at!

3 A TOUR OF GKLEE-MODE

This section will go over the features of Gklee-Mode, including what is happening under the hood, the output that is generated, and the user controls that are provided in the form of key bindings and mouse aware buffers.

3.1 Compilation

This subsection is to describe the general work flow that one uses to analyze their program with Gklee-Mode.

First, the user must open the C or C++ file containing the driver/host code and including a *main* function. *Currently, Gklee-Mode isn't set up to compile multiple file projects.* To work around this issue, you may use *#include* statements in your main file to 'pull in' the ancillary files.

Next, while the focus within Emacs is in the main source file, the user simply executes *run-gklee*. This is available through a key binding, *[meta]-gr* and then choosing the optimization level to set the LLVM compiler output to.

Once *run-gklee* has been initiated, the code will be compiled into LLVM bytecode, using the chosen optimization level and also with the option *-g*, which causes debugging information to be embedded into the generated object code. This is necessary for generating a trace and to match the trace to detected errors later.

If there is an error in compilation, a message will be generated and displayed in the mini-buffer. To see what is happening with the compilation, a user may toggle an Emacs window to the **gklee-compile-debug** window. This shows the command line used to invoke llvm-g++, and the output generated by this.

3.2 Concolic Execution

After the compilation completes successfully, GKLEE is executed, passing on its command line all of the options selected and the name of the object file generated in the previous step. The main result buffer, **gklee-run**, displays the number of program executions explored (one for each generated path, depending upon the user's choice of setting program variables to symbolic). Also, all of the output of GKLEE is dumped into the **gklee-run-debug** buffer. If the execution of GKLEE is unsuccessful, a message to this effect is posted in the mini-buffer.

3.3 Trace Walking and Debugging

Once GKLEE successfully completes execution, the **gklee-run** buffer is populated with some general statistics along with a section for each generated path. They include metrics on CUDA performance issues, and will list any detected anomalies by description (in red) and information on the location of the problem in the user code.

The trace synopses in **gklee-run** are sensitive to the mouse. Each trace is highlighted when the mouse floats over it; also, the user may click on the trace in general or on a specific issue. This opens a trace window for that execution, and if an error was clicked on, will set the focus to the first trace line in question (also highlighted in red).

Each trace line in the trace buffer / window is mouse aware. By clicking on a line, the related line in the user source code is brought to focus in that buffer, or if the file containing that instruction isn't open, it will be opened and scrolled to the correct location. There is also an arrow in the right margin that points to the related source code instruction.

When the user has Emacs focus set to the trace buffer there are a number of features available for navigation and narrowing the amount of information presented. With the correct keystrokes (see §4), the user can filter out unwanted CUDA blocks, threads and warps, and also limit visible trace lines to certain locations in code, or by entire source code files. Also, by highlighting sections of the code in a source buffer, these regions can be selected for specific study in the trace. Another feature is the ability to toggle between having a view of the LLVM bytecode in the trace or not. When viewing bytecode, there are many more instructions as source level instructions often generate many intermediate level instructions.

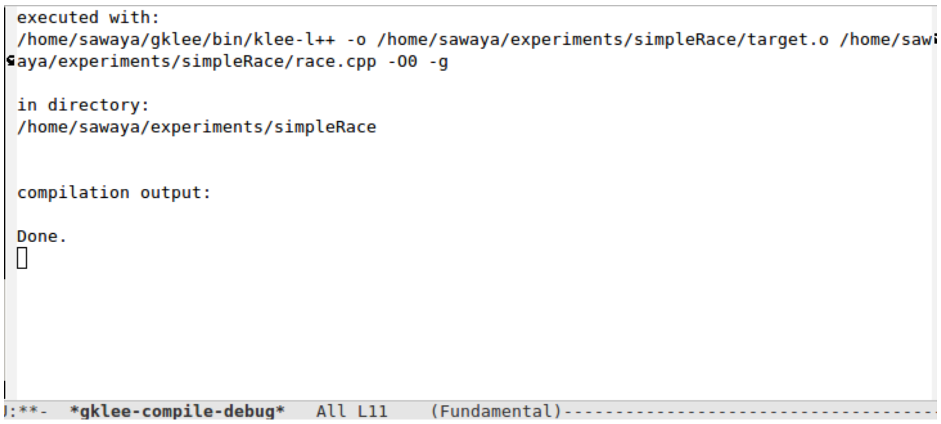
Another feature available from the trace buffer is the ability to invoke a stepping mode (§4). When started, the first unfiltered line in the trace is highlighted, and a key is available to step to the next unfiltered instruction. When stepping to the next trace line, the related source code line is also shown with the 'next instruction' arrow.

These features allow the CUDA developer to locate problems in their code, set focus to particular features of their program, such as a section of code and / or a particular warp, and the steps leading up to an error or performance issue.

3.4 GKLEE Buffers

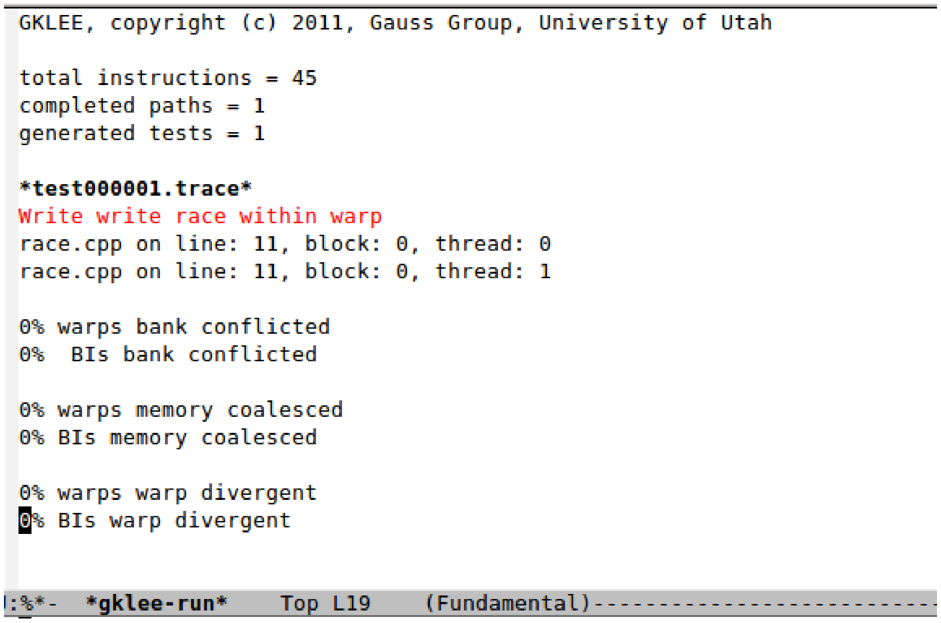
This section, along with §4 explains how the user can control GKLEE and modify the information that is available, and how information is displayed. Information is displayed in Emacs buffers and the mini-buffer, with much of it sensitive to mouse actions. Mouse clicks and key bindings are provided so the user may modify the behavior of GKLEE and Gklee-Mode.
Output from GKLEE and Gklee-Mode is provided by using Emacs buffers. There is some static data, and many entries in these buffers are mouse-clickable in order to drill down into more information, or toggle settings.
Here is a list of the buffers, what stage in the general process they are relevant, and what controls they provide.

- Buffer **gklee-compile-debug** contains the output of the LLVM compilation and its command line. See Figure 2
- Buffer **gklee-run** displays an overview of the results of the GKLEE execution. It includes a synopsis on each generated trace. Each trace and issue/error within them is navigable with a mouse click. See Figure 3



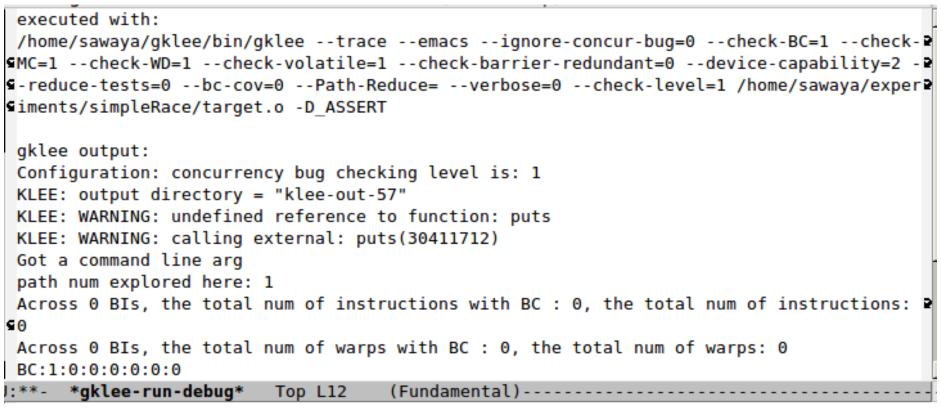
```
executed with:
/home/sawaya/gklee/bin/klee-l++ -o /home/sawaya/experiments/simpleRace/target.o /home/saw
aya/experiments/simpleRace/race.cpp -O0 -g

in directory:
/home/sawaya/experiments/simpleRace


compilation output:

Done.



-:**-  *gklee-compile-debug*   All L11    (Fundamental)-----------------------------------
```

Figure 2: The *gklee-compile-debug* buffer after compilation



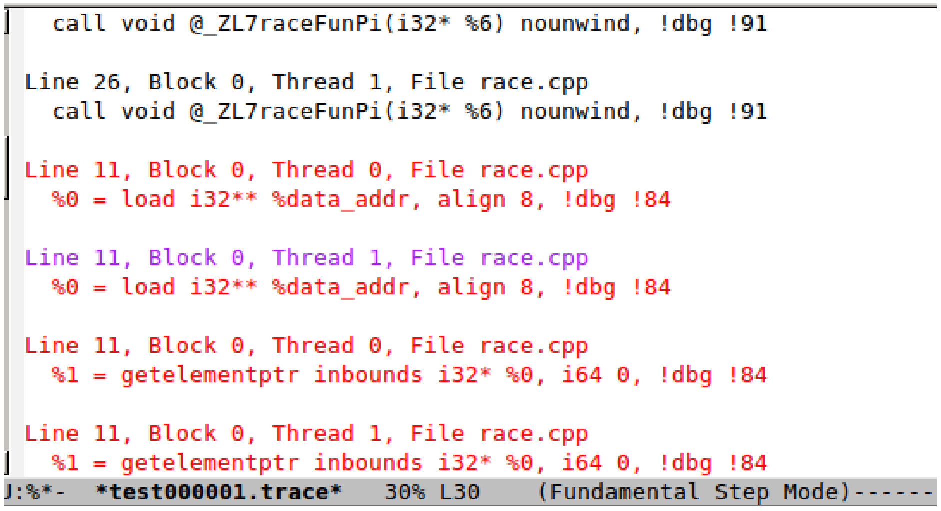
```
GKLEE, copyright (c) 2011, Gauss Group, University of Utah

total instructions = 45
completed paths = 1
generated tests = 1

*test000001.trace*
Write write race within warp
race.cpp on line: 11, block: 0, thread: 0
race.cpp on line: 11, block: 0, thread: 1

0% warps bank conflicted
0%  BIs bank conflicted

0% warps memory coalesced
0% BIs memory coalesced

0% warps warp divergent
0% BIs warp divergent


-:%*-  *gklee-run*     Top L19    (Fundamental)--------------------------
```

Figure 3: The *gklee-run* buffer



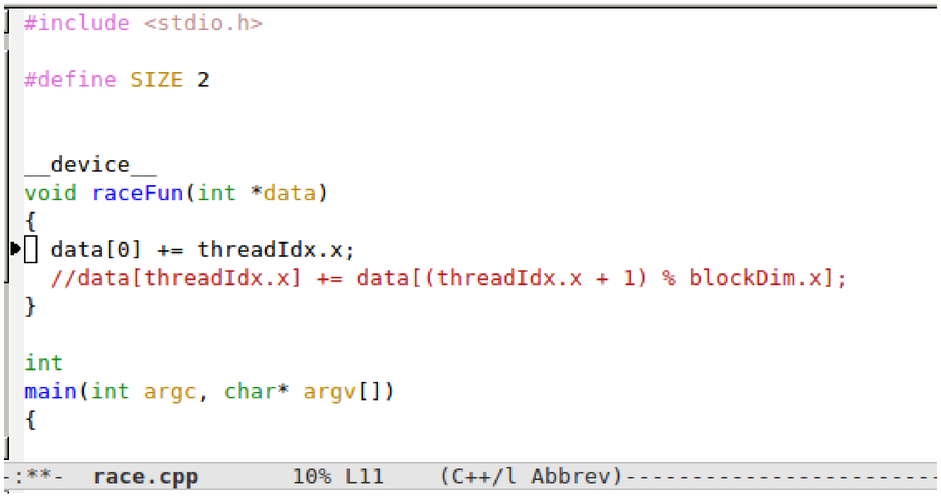
```
executed with:
/home/sawaya/gklee/bin/gklee --trace --emacs --ignore-concur-bug=0 --check-BC=1 --check-
MC=1 --check-WD=1 --check-volatile=1 --check-barrier-redundant=0 --device-capability=2 -
-reduce-tests=0 --bc-cov=0 --Path-Reduce= --verbose=0 --check-level=1 /home/sawaya/exper
iments/simpleRace/target.o -D_ASSERT

gklee output:
Configuration: concurrency bug checking level is: 1
KLEE: output directory = "klee-out-57"
KLEE: WARNING: undefined reference to function: puts
KLEE: WARNING: calling external: puts(30411712)
Got a command line arg
path num explored here: 1
Across 0 BIs, the total num of instructions with BC : 0, the total num of instructions:
0
Across 0 BIs, the total num of warps with BC : 0, the total num of warps: 0
BC:1:0:0:0:0:0:0
U:**-  *gklee-run-debug*   Top L12    (Fundamental)-------------------------------------
```

Figure 4: The **gklee-run-debug** buffer



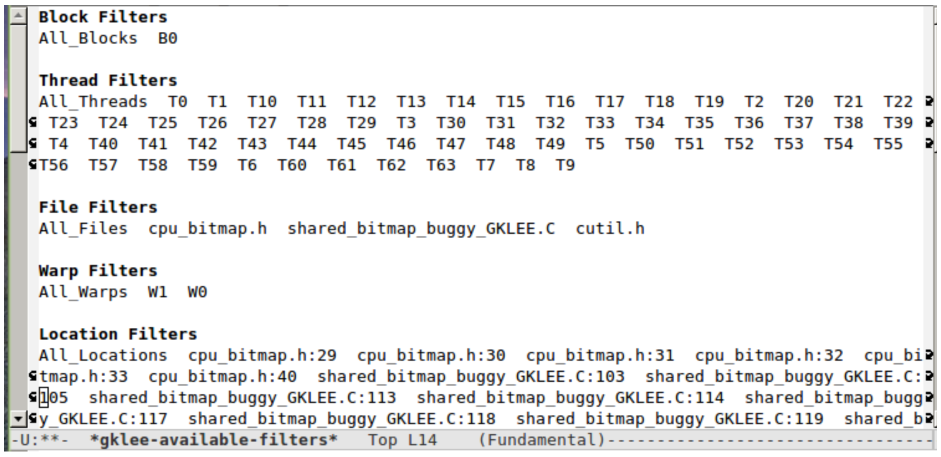
```
   call void @_ZL7raceFunPi(i32* %6) nounwind, !dbg !91

 Line 26, Block 0, Thread 1, File race.cpp
   call void @_ZL7raceFunPi(i32* %6) nounwind, !dbg !91

 Line 11, Block 0, Thread 0, File race.cpp
   %0 = load i32** %data_addr, align 8, !dbg !84

 Line 11, Block 0, Thread 1, File race.cpp
   %0 = load i32** %data_addr, align 8, !dbg !84

 Line 11, Block 0, Thread 0, File race.cpp
   %1 = getelementptr inbounds i32* %0, i64 0, !dbg !84

 Line 11, Block 0, Thread 1, File race.cpp
   %1 = getelementptr inbounds i32* %0, i64 0, !dbg !84
U:%*-  *test000001.trace*   30% L30    (Fundamental Step Mode)------
```

Figure 5: The trace buffer



```
#include <stdio.h>

#define SIZE 2


__device__
void raceFun(int *data)
{
  data[0] += threadIdx.x;
  //data[threadIdx.x] += data[(threadIdx.x + 1) % blockDim.x];
}

int
main(int argc, char* argv[])
{

-:**-  race.cpp       10% L11     (C++/l Abbrev)------------------------
```

Figure 6: A source buffer, with the trace enabled



```
Block Filters
All_Blocks  B0

Thread Filters
All_Threads  T0  T1  T10  T11  T12  T13  T14  T15  T16  T17  T18  T19  T2  T20  T21  T22
 T23  T24  T25  T26  T27  T28  T29  T3  T30  T31  T32  T33  T34  T35  T36  T37  T38  T39
 T4  T40  T41  T42  T43  T44  T45  T46  T47  T48  T49  T5  T50  T51  T52  T53  T54  T55
T56  T57  T58  T59  T6  T60  T61  T62  T63  T7  T8  T9

File Filters
All_Files  cpu_bitmap.h  shared_bitmap_buggy_GKLEE.C  cutil.h

Warp Filters
All_Warps  W1  W0

Location Filters
All_Locations  cpu_bitmap.h:29  cpu_bitmap.h:30  cpu_bitmap.h:31  cpu_bitmap.h:32  cpu_bi
tmap.h:33  cpu_bitmap.h:40  shared_bitmap_buggy_GKLEE.C:103  shared_bitmap_buggy_GKLEE.C:
105  shared_bitmap_buggy_GKLEE.C:113  shared_bitmap_buggy_GKLEE.C:114  shared_bitmap_bugg
y_GKLEE.C:117  shared_bitmap_buggy_GKLEE.C:118  shared_bitmap_buggy_GKLEE.C:119  shared_b
-U:**-  *gklee-available-filters*   Top L14    (Fundamental)-------------------------------
```

Figure 7: A filter buffer (one of two)

- Buffer **gklee-run-debug** contains all of the output generated by GKLEE. It is useful to understand how GKLEE works and to see what went wrong on an unsuccessful execution. See Figure 4

- The trace buffer is populated when a trace item is clicked on in the **gklee-run** buffer. The trace contains a line for each low-level instruction (when viewing LLVM is toggled, or one per source instruction otherwise) and is sensitive to mouse clicks. See §4 for bindings available from the trace buffer. Also, from within the trace buffer you may activate stepping mode. See figure 5

- The source buffer is like an ordinary C/C++ buffer in Emacs, unless a trace line has been selected in the trace buffer, either through mouse-click, keyboard or through stepping mode. When a trace line has been activated, the buffer window will be centered on the related instruction, and there will be an arrow in the right margin pointing to that instruction. *Also*, there is a context menu available from source code buffers that allows the user to limit which instruction locations are visible in the trace and for stepping; also stepping mode may be activated from here. See figure 6

- The filter buffers (there are two: **gklee-available-filters** and **gklee-active-filters*) are used to choose trace items to view or ignore, and to see which are visible and which are not. Criteria available are blocks, threads, files, warps and locations. By clicking on an item in the **gklee-available-filters* buffer, that item will be filtered out of the trace window and the excluded trace lines won't be activated in step mode. The reverse is also true, as the user may reactivate trace items by clicking on an identifier in the **gklee-active-filter** window. Also, when filter criteria are modified by using a source buffer context window, the filter buffers are updated accordingly. Refer to figure 7 for an example view

4 QUICK REFERENCE

This section gives a listing of available global and buffer specific key bindings. By using the key bindings in Gklee-Mode, all GKLEE specific options are configurable and also the features of Gklee-Mode are exposed. Please refer to the GKLEE manual (or ask us) for complete descriptions of the GKLEE features that can be controlled here. Keep in mind that **M** indicates the Emacs *Meta* key (you may visit the Emacs wiki for system specific key bindings: `http://www.emacswiki.org/emacs/`

Here is a list of the key bindings globally available in Gklee-Mode:

M-gr	GKLEE run (will prompt for optimization level)
M-gk	GKLEE kill (stops GKLEE execution)
M-gtcb	Toggle the –ignore-concur-bug flag passed to GKLEE
M-gtbc	Toggle the –check-BC flag passed to GKLEE
M-gtmc	Toggle the –check-MC flag passed to GKLEE
M-gtwd	Toggle the –check-WD flag passed to GKLEE
M-gtcv	Toggle the –check-volatile flag passed to GKLEE
M-gsdc	Pass the flag –device-capability `<n>` to GKLEE
M-gtrt	Toggle the –reduce-redundant-tests flag passed to GKLEE
M-gtgc	Toggle the –bc-cov flag passed to GKLEE
M-gspr	Pass the flag –reduce-path with 'b' or 't' or none to GKLEE
M-gtv	Toggle the –verbose flag passed to GKLEE
M-gscl	Pass the flag –check-level flag to GKLEE with an argument of 1/2/3
M-gupa	Helps pass arguments to the user program being run under GKLEE
M-guca	Helps pass arguments to `klee-l++`
M-gaga	Helps pass arguments to GKLEE (e.g., –max-time etc)

The meaning of the above GKLEE flags is as follows:

–ignore-concur-bug	If set, no global and shared memory race conditions are checked, <0(default)/1>
–generate-perform-tests	If set, test inputs leading to performance defects are generated, <0(default)/1>
–check-BC	Check bank conflicts, <0/1(default)>
–check-MC	Check whether global memory accesses can be coalesced, <0/1(default)>
–check-WD	Check whether there exists warp divergence, <0/1(default)>
–check-volatile	Check whether volatile keywork is missing, <0/1(default)>
–device-capability	Set device capability (0): 1.0-1.1; (1): 1.2-1.3; (2): 2.x (default)
–reduce-redundant-tests	outputs only a subset of test cases <0(default)/1>
–bc-cov	calculate bytecode coverage for the threads <0(default)/1>
–reduce-path	path reduction <"B/T">
–verbose	Dump informative debugging information <0(default)/1>
–check-level	Race check level (0): no check; (1): shared memory (2): shared & global memory (default)

This is a list of the key bindings available from the trace buffer:

backspace	Exit trace
M-gta	Toggle show/hide of LLVM level instructions
M-gss	Enter stepping mode
M-gne	Set focus of the trace buffer to the next error

Here are the key bindings when in *gklee step mode*:

n	next step
q	quit step mode
c	continue – will execute until error instruction

5 OBTAINING GKLEE, INSTALLING

There are a few ways that you can obtain GKLEE for yourself and run it. You may:

- run GKLEE on our servers and then analyze the results with your Emacs (go to `http://www.cs.utah.edu/fv/GKLEE/run/`)
- Use the Ubuntu ISO – you can use this as a virtual machine in some environment such as VMWare Workstation or VirtualBox
- Get the latest version through our Subversion code repository

With the first item you may create a session on our server, upload your code, and then analyze it with the click of a button. When it is complete, press another button to transfer the results to a location of your choice. You may then load the results into your local Emacs for exploration.

If you are using the current code base, you can find directions on configuring and installing GKLEE in our wiki, located at `http://www.cs.utah.edu/fv/mediawiki/index.php?title=GKLEE`.

6 ANOTHER TUTORIAL EXAMPLE

For this tutorial, we will take an example that is included with GKLEE, (and coded from an example in the book CUDA by Example) called 'Shared Bitmap', and analyze it with GKLEE. We will then choose an issue to explore, filtering out irrelevant data and will then step through the trace to understand the origins of the issue.

To begin, navigate to the top level of your GKLEE installation (.../gklee) and proceed to (i.e. cd) .../gklee/examples/CBE/Chapter5_shared_bitmap. This example is contained in a single source file,
shared_bitmap_correct_GKLEE.C.

Open this file by issuing 'emacs shared_bitmap_correct_GKLEE.C' on the command line.

Now you should have Emacs open with the combined host / kernel code for the shared bitmap example.

Now you are ready to invoke the process of compilation and analysis by GKLEE. To begin, enter the sequence '**M**-gr', or 'gklee-run'. You will be prompted for the optimization level; enter '3' (which will pass the option '-O3' to the LLVM compiler).

The first thing you will notice is that Emacs split its windows so that you now have four windows showing. The top left should contain the source buffer, the top right the **gklee-run** buffer, the bottom left displays the **gklee-run-debug** buffer and the bottom right the **gklee-compile-debug* buffer.

If compilation doesn't fail (you can tell by looking at the **gklee-compile-debug* buffer), the **gklee-run** buffer will contain the results of the GKLEE execution as it progresses. You will see it count the number of paths completed and then, once the GKLEE execution is done, you will see a list of information for **test000001.trace**, the path generated for this program.

If the compilation does fail for some reason, you can find out why by examining the contents of **gklee-compile-debug**.

If the execution of GKLEE fails, there will be a message in the mini-buffer to that effect, along with the full output of GKLEE in **gklee-run-debug** to help diagnose the issue.

This example contains a long list of bank conflicts. Bank conflicting shared memory accesses are a performance issue in CUDA programs where concurrently executing instructions (within a warp) are requesting a read or write from shared memory from locations served by the same memory bank. However, since the span of the addresses exceed the width of a memory bank (as of CUDA 4.0 banks serve 128 byte segments), the shared memory reads or writes have to be serialized instead of being issued concurrently.

Now we will choose one of the bank conflicts listed in **gklee-run**. Enter the key sequence '**Control**-s (for *search*), and type 'write'. This will expose the first Write write bank conflict detected in the program, which should be on line 69, block 3 and threads 251 and 255. Click somewhere in the area close to 'Write write bank conflict' and the two lines below that list the location information for each thread. This will cause the top right window to display the trace buffer, in this case **test000001.trace**, with the first error location brought to focus. Also, you will see the filter windows, **gklee-available-filter** and **gklee-active-filters** displayed in the two bottom windows.

The line in focus in the trace buffer's window is the first instruction that may be involved in the indicated write write bank conflict. We say that it may be involved because trace lines are associated with the issue only by location in code, thread id, and block id. It is possible that this location was executed by these processing elements more than once writing to non-conflicting addresses due to a loop. But we know that the issue did occur at least once, and we make it easy to view all of the occurrences. In future work the exact occurrence could be identified by an instruction trace counter or some other means. We will cycle through all the occurrences of this issue below with a Gklee-Mode feature.

To see all the potential occurrences of this issue, let's begin by removing some information to make it easier to understand. The first thing is that we have one trace line per intermediate instruction (LLVM), and in the case of this source instruction, has many of them. Also, since ultimately the machine level instructions are executed in lock step within a warp (so we execute the intermediate instructions inter-warp in lockstep), you don't see all the intermediate instructions that compose the source instruction for this processing element (block and thread).

Click on the line in focus in the trace window, 'Line 79, Block 3, Thread 251 . . .'. You will see the corresponding source instruction in the source code buffer window. You can see that this is a rather 'compound' source instruction, as it involves loads of processing element identification constants ('threadIdx.x' and 'threadIdx.y'), two variables, some complex mathematics for the value to be assigned to shared memory, etc. Now we will hide the intermediate instructions in the trace. Just enter the sequence '**M**-gta', for 'toggle assembly', and you will see the trace collapse to only location / processing element identifiers.

To make it easier to examine the circumstances in the program execution leading up to this issue, let's filter out the threads that are not involved in it. To begin with, we will filter out all of the irrelevant cooperative thread arrays, or blocks, by clicking 'All_Blocks' in the **gklee-available-filters* buffer. When you do this, you should find that there are no longer any trace lines visible in the trace window, and inside the **gklee-active-filters** buffer's window underneath **Block_Filters** shows 'All_Blocks B0 B1 B2 B3. We will now show only the block of interest, 'B3', by clicking on it. Now you will see that 'B3' is back in the available filter window, and the trace window shows only instructions executed by block 3.

Now let's narrow our trace view even further by showing only the threads of interest, 'T251' and 'T255'. To do this, again we will filter them all and then unfilter the ones we want to see. Click 'All_Threads' to filter out all threads, and then click on 'T251' and 'T255' in **gklee-active-filters* to reveal them once again.

One more step to limit the view. We will now display only the source locations that we are interested in studying. The source buffer, *shared_bitmap_correct_GKLEE.C* should still be pointing to line 69, the location of the write write shared memory bank conflict. Let's limit our study to the instructions leading up to this instruction within this function. To do so, using the mouse left button, highlight the lines in the source buffer from 55 to 70, getting the function signature and the currently focused line. Now, by right clicking on the highlighted region, you will be presented with a context menu that exposes a few functions of Gklee-Mode. Click on 'Filter other lines out'. You will see that all the other locations in **gklee-available-filters** are moved to **gklee-active-filters*, and only the trace lines of interest remain in the trace window.

Finally, we will use the automatic stepping mode to go through the section of execution we are studying. To begin, click on a trace item to be sure that the input focus is set to that window. Then, if you enter the key sequence '**M**-gss', you will enter stepping mode.

In order to understand this bank conflict, the processing element's identification must be translated into a two dimensional id (as this example uses a block size of 8 x 8 and a grid size of 2 x 2). Gklee grants each thread a sequential id (and uses the same scheme to identify blocks). Since we are studying threads 251 and 255, we must calculate the block and thread ids for the x and y dimensions. They are assigned in an x-dimension major order.

Here are the formulas to extract the 2 dimensional identifiers from the

1-dimensional block and thread ids (using *tid*, *bid*, *bdim* and *ltid* [the block local tid] for the 1-d identifiers, and the CUDA built-in processing element identifier variable names for 2-d)[2]:

$$bdim = blockDim.x \times blockDim.y$$

$$bid = \frac{tid - tid \bmod bdim}{bdim}$$

$$ltid = tid - bid \times bdim$$

$$blockIdx.x = bid \bmod gridDim.x$$

$$blockIdx.y = \frac{bid - bid \bmod gridDim.x}{gridDim.x}$$

$$threadIdx.x = ltid \bmod blockDim.x$$

$$threadIdx.y = \frac{ltid - ltid \bmod blockDim.x}{blockDim.x}$$

In the case of $bid = 251$ and $bid = 255$, the values are assigned
$threadIdx.x = 3$ and $threadIdx.y = 7$ and
$threadIdx.x = 7$ and $threadIdx.y = 7$ respectively.

So now Gklee-Mode is configured to step through the kernel up to the point of interest (the write write bank conflict), showing only the two threads of interest, for $bid = 251$ and $bid = 255$. Simply press n to cycle to the next instruction.

As you step through, you will see that the array *shared* is accessed at
$shared[threadIdx.x][threadIdx.y]$ which, for $bid = 251$, is equivalent to
$shared[7 + 3 * blockDim.x = 31]$, and for $bid = 255$ is
$shared[7+7*blockDim.x = 63]$. This means that the accesses are $63-31 = 32$ floats apart (which are currently 4 bytes each). There are 32 shared memory banks (under CUDA device capability 2.0), each of which are assigned

[2]Of course, we already know bid from GKLEE

addresses 4 bytes apart. This means that addresses that are 128 bytes apart (or 32 words)0 are served by the same memory bank.

7 CONCLUDING REMARKS

This document gave you an overview of the interface Gklee-Mode, which is an Emacs extension that allows the CUDA developer to easily analyze their program with the GKLEE tool. It provides a controlled environment that sets all available options for GKLEE, performs all necessary compilation and command line executions, and connects the provided analysis directly to the developer's CUDA source code. This way, users can select a trace with some interesting feature (such as a correctness or performance issue), filter out superfluous execution elements (such as threads, blocks, warps and code locations) and step directly to the event under study.

Please refer to the GKLEE website for further information:
`http://www.cs.utah.edu/fv/GKLEE/`

References

[1] CUDA Programming Guide Version 4.0. `http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf`.

[2] C. Cadar, P. Godefroid, S. Khurshid, C. Pasareanu, K. Sen, N. Tillmann, and W. Visser. Symbolic Execution for Software Testing in Practice – Preliminary Assessment. In *Proc. Impact Project Focus Area in 33rd International Conference on Software Engineering (ICSE'11)*, pages 1066–1071, 2011.

[3] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI, 8th USENIX Symposium*, 2008.