A Short Tutorial of CKLEE for CUDA (Version 0.01) [*]

Guodong Li

November 3, 2010

1 Installation

This tool has been installed at /uusoc/facility/res/uv/ligd/klee-FLA. The installation is based on 64-bit Ubuntu 10.04, which is the default OS at machine shell.cs.utah.edu. This means that you shall be able to run it at shell.cs.utah.edu without recompilation.

The main directory contains the following directories:

- **llvm-2.6.** Source and executable files of LLVM v2.6.
- **llvm-gcc-4.2-2.6.** Executable files of LLVM-GCC 4.2.
- **Cklee.** Source and executable files of CKLEE.
- **CUDA.** Examples (*e.g.* tutorial examples).
- **bugbench1.1.** A benchmark by Univ. of Wisconxin, which contains well-known bugged C programs.

2 Running CKLEE

First of all, modify your bash.rc or csh.rc to

- add $(FLA_KLEE_HOME_DIR)/klee/Release/bin, $(FLA_KLEE_HOME_DIR)/llvm-2.6/Release/bin, $(FLA_KLEE_HOME_DIR)/llvm-gcc-4.2-2.6/bin and $(FLA_KLEE_DIR)/bin to the environment variable PATH; and
- set environment variable FLA_KLEE_HOME_DIR to be the path of CKLEE if you haven't done so.

Here is what my .bashrc looks like:

```
export PATH=/home/gli/klee-FLA/klee/Release/bin:/home/gli/klee-FLA/llvm-2.6/Release/bin:
           /home/gli/klee-FLA/llvm-gcc-4.2-2.6/bin:/home/gli/klee-FLA/bin:$PATH
export FLA_KLEE_HOME_DIR=/home/gli/klee-FLA
```

An example .cshrc is:

```
setenv FLA_KLEE_HOME_DIR /uusoc/facility/res/uv/ligd/klee-FLA
setenv PATH ${PATH}:${FLA_KLEE_HOME_DIR}/Cklee/Release/bin:
           ${FLA_KLEE_HOME_DIR}/llvm-2.6/Release/bin:
           ${FLA_KLEE_HOME_DIR}/llvm-gcc-4.2-2.6/bin:${FLA_KLEE_HOME_DIR}/bin
```

[*]This tutorial is for the 64-bit linux version.

Table 1: Commonly used CKLEE commands.

klee-lcc	compile C programs into LLVM bytecode
klee-gcc	compile C program into machine code (for real machines) with the information for gcov
cklee	symbolically execute a bytecode in CKLEE
klee-show-tests	show all the generated test cases
klee-replay-tests	replay all the test cases automatically and generate coverage information

Then, for a C program foo.cpp, use klee-l foo.cpp to compile it into bytecode, then use cklee foo.o to invoke CKLEE on it. Other commands (see Table 1) are available to track the results. Most of these commands are simply shell scripts. For instance, command cklee++ is defined as follows.

```
   cklee --libc=uclibc $*
```

It links the uClibc library when CKLEE is running. Note that we put the C library into one single file klee-FLA/klee/klee-uclibc/lib/libc.a. You may use llvm-ar t libc.a to show its content:

```
...
libc/unistd/usershell.os
libc/unistd/usleep.os
```

As another example, klee–l++ is actually the following command. It uses the uClibc (rather than your default GCC headers) when the source program is compiled into byte code. If you use other headers, then the generated bytecode might be not compatible with the C library CKLEE will link, leading to a runtime error.

```
llvm-g++ -emit-llvm -O3 -I $FLA_KLEE_HOME_DIR/klee/include/klee -I ./
         -I $FLA_KLEE_HOME_DIR/klee/klee-uclibc/include -c $*
```

3 Simple Examples

The "CUDA/tutorial" directory contains examples on how to use CKLEE to symbolically execute and replay CUDA kernel programs. Let's go through them one by one. In general, the order of processing kernel "foo" is:

1. Write a drive program "foo.C" to indicate how the kernel is invoked.
2. Use "klee-lcc foo.C" to compile the source into LLVM bytecode sort.o.
3. Use "cklee foo.o" to symbolically execute the bytecode; CKLEE will generate test cases. You will need to specify the block and grid sizes in this phase.
4. Use "klee-show-tests" to obtain the generated test cases.
5. Use "klee-g++ foo.C" to compile the source (you may first delete the existing foo.gcda), then run "klee-replay-tests" to replay all the test cases.
6. Then, in order to obtain the coverage information, type "gcov -b foo.C" and you will see the statistics.

Note: you will need to copy the "CUDA/tutorial" to your own place since this directory is not writable.

3.1 Driver Program Format

CKLEE expects a drive program of the following format. Function "__begin_GPU()" marks the start of kernel execution; and function "__end_GPU()" marks the end of kernel execution. CUDA's kernel invocation syntax <<< ... >>>(kernel) can be converted into "__begin_GPU(); kernel(); __end_GPU();" with the configuration information (see the next section for how to set the configuration in the command line).

```
int main() {

  ... // CPU code

  __begin_GPU();
  ... // GPU kernel code
  __end_GPU();

  ... // CPU code
}
```

The tutorial examples show how to write such drivers. Note that CKLEE supports both CPU and GPU code (and their combination); and you may use it purely for executing CPU programs.

3.2 CUDA Configuration: test_config.c

This example prints out the configuration information (*e.g.* grid size, block size, the threadIdx and blockIdx of each thread). The first step is to use "klee-lcc test_config.c" to produce the bytecode "test_config.o". Then we use "cklee test_config.o" to run CKLEE on the bytecode. The output is:

```
GridDim = <1, 1>
BlockDim = <2, 1, 1>

CKLEE: Start executing a GPU kernel

threadIdx = <0, 0, 0>
blockIdx = <0, 0>
threadIdx = <1, 0, 0>

CKLEE: Finish executing a GPU kernel
```

This indicates that the grid size is $< 1, 1 >$ and the block size is $< 2, 1, 1 >$. Obviously the number of threads is 2 in this configuration. The threadIdx of thread 1 is $< 0, 0, 0 >$; and that of thread 2 is $< 1, 0, 0 >$. That is, for thread 1, threadIdx.x = 0; and for thread 2, threadIdx.x = 1. This is the default GPU configuration.

Let's try another configuration. The following command (here I assume you are using the c shell rather than the bash shell)

```
  cklee --blocksize=\[3,2,1\] --gridsize=\[2,3\] test_config.o
```

specifies that the block size is $< 3, 2, 1 >$ and the grid size is $< 2, 3 >$. We will have 2*3=6 blocks and 3*2*1*6 = 36 threads. The output shows the threadIdx of thread 0, 1, ..., 35.

```
GridDim = <2, 3>
BlockDim = <3, 2, 1>

CKLEE: Start executing a GPU kernel

threadIdx = <0, 0, 0>
blockIdx = <0, 0>
threadIdx = <1, 0, 0>
threadIdx = <2, 0, 0>
threadIdx = <0, 1, 0>
threadIdx = <1, 1, 0>
threadIdx = <2, 1, 0>
threadIdx = <0, 0, 0>
blockIdx = <1, 0>
threadIdx = <1, 0, 0>
threadIdx = <2, 0, 0>
```

3.3 Deadlock and Race Checking: test_barrier_race.c

This example has a kernel which contains deadlocks, and a kernel which will lead to races. The deadlock bug is due to barrier mismatch. To see this, compile the program for deadlock checking:

```
  klee-lcc  test_barrier_race.c
```

You may use only 2 threads to generate the bug:

```
cklee test_barrier_race.o

...

CKLEE: Start executing a GPU kernel

KLEE: ERROR: execution halts on a barrier mismatch, which will incur a deadlock
KLEE: NOTE: now ignoring this error at this location
```

Another bug is about shared-memory races. You can use the following command to produce bytecode for race checking:

```
  klee-lcc  test_barrier_race.c -DRACE
```

Let's try 4 threads

```
  cklee --blocksize=\[4\] test_barrier_race.o
```

which outputs

```
CKLEE: Start executing a GPU kernel

CKLEE: Threads 0 and 3 incur a W-R race on address 19153648
KLEE: ERROR: execution halts on encountering a race
```

A write-read race occurs between thread 0 and thread 3. You can use KLEE to replay the cause of this bug (*i.e.* replay the real execution leading to this bug). But we won't go into this in this tutorial.

We may want to get more information about this bug. By using

```
  cklee --verbose=1  --blocksize=\[4\] test_barrier_race.o
```

we can see more debugging message. It shows how threads evolve. As in PUG, the ReadSet and WriteSet are used to determine whether there exist races.

```
CKLEE: Start executing a GPU kernel

Thread 0 reaches a barrier: moving to the next thread.
Thread 1 reaches a barrier: moving to the next thread.
Thread 2 reaches a barrier: moving to the next thread.
Thread 3 reaches a barrier: moving to the next thread.

Start checking races at SharedMemory 0...
Read Set:
<43409600, t3> <43409604, t0> <43409608, t1> <43409612, t2>
Write Set:
<43409600, t0> <43409604, t1> <43409608, t2> <43409612, t3>

CKLEE: Threads 0 and 3 incur a W-R race on address 43409600
KLEE: ERROR: execution halts on encountering a race
```

You can also check whether there are races on the device memory and the CPU memory by setting the checklevel value. Note that the accesses to these memories won't be synchronized by the "__syncthreads()" call. In fact we perform the check when we encounter the "__end_CPU()" function.

```
  cklee --checklevel=2  --blocksize=\[4\] test_barrier_race.o
```

When the value of checklevel is 0, we don't check any concurrency bug; when it is 1, we check deadlocks and shared memory races; when it is 2, we also check races in the device memory and the cpu memory (which can be called inter-block races). Note that all the command-line options can be used in the symbolic execution.

3.4 Symbolic Execution: bitonic.C

Program "bitonic.C" contains code for both concrete and symbolic executions. Let's try the concrete execution first, where a concrete value is given to the input array values:

```
__input__ unsigned values[NUM] = {6, 5, 2, 1, 4, 3};
```

As usual, first compile it into bytecode:

```
 klee-lcc bitonic.C
```

Then we try the case of 6 threads:

```
 cklee --blocksize=\[6\] bitonic.o
```

The output indicates that this kernel is incorrect when the number of threads is 6 (recall the PUG tells us the number of threads must be the power of 2). Hence CKLEE can be used to perform concrete execution and find bugs.

```
CKLEE: Finish executing a GPU kernel

Output values:
1 2 5 6 4 3
The sorting algorithm is incorrect since values[4] < values[3]!
```

Now we try the most interesting part: symbolic execution. We will tell the compiler to pick the code for symbolic execution:

```
 klee-lcc bitonic.C -D_SYM
```

Then we try the case of 4 threads:

```
 cklee --blocksize=\[4\] bitonic.o
```

It passed the sanity check, *i.e.* the postcondition requiring that the output array is sorted even the inputs are symbolic. It will produce 28 test cases (corresponding to 28 paths).

```
...
KLEE: done: total instructions = 4411
KLEE: done: completed paths = 28
KLEE: done: generated tests = 28
```

We can use "klee-show-tests" to see all these test cases:

```
ktest file : 'klee-last/test000001.ktest'
args       : ['bitonic.o']
num objects: 1
object    0: name: 'input'
object    0: size: 16
object    0: data: '\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'

ktest file : 'klee-last/test000002.ktest'
args       : ['bitonic.o']
num objects: 1
object    0: name: 'input'
object    0: size: 16
object    0: data: '\x00\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00'

...
```

A natural question is: how to use these test cases? A possible way is to replay them in the real setting. We can first compile the driver program using gcc (the replaying won't use CKLEE's symbolic executor anymore).

```
 cklee-gcc bitonic.C -D_SYM -o sym.out
```

This will generate an object file for replaying. Then the following command

```
klee-replay-tests sym.out
```

will display

```
Running test: klee-last/test000001.ktest
Running test: klee-last/test000002.ktest
...
Running test: klee-last/test0000028.ktest
```

By using "gcov -b bitonic.C" we have

```
File 'bitonic.C'
Lines executed:48.28% of 29
Branches executed:25.00% of 16
Taken at least once:12.50% of 16
Calls executed:83.33% of 6
```

Oops! These numbers are not what we want! This is because we use CPU to replay the kernel, where only thread 0 is executed. If you have CUDA hardware, you can use NVCC to compile the driver program and run the kernel (however I am not sure whether gcov will work in this case).

To enforce all the threads to participate in the replaying, I write a Perl script to convert the kernel to a version where the CPU simulates the execution of all the threads. It generates file "bitonic.cpp".

```
./convert.pl bitonic.cc
```

Now using this file to do the replaying with the generated test cases:

```
 cklee-gcc bitonic.cpp -D_SYM -o sym.out
 klee-replay-tests sym.out
```

It displays the information ensuring that the output array is sorted:

```
...
Input values:
7 6 0 4
Output values:
0 4 6 7
```

Now, the coverage information (obtained by using "gcov -b bitonic.cc") is:

```
File 'bitonic.cpp'
Lines executed:100.00% of 30
Branches executed:100.00% of 22
Taken at least once:100.00% of 22
Calls executed:100.00% of 5
bitonic.cpp:creating 'bitonic.cpp.gcov'
```

3.5 Optimizations: bitonic.C

The number of test cases may blow up when the number of threads is large. Take bitonic.C for example, 28 test cases are produced for 4 threads. We can reduce this number without sacrifying too much coverage. Command line option "reduce-tests" will try to avoid duplicate test cases.

```
cklee --blocksize=\[4\] --reduce-tests bitonic.o

...
KLEE: done: total instructions = 4411
KLEE: done: completed paths = 28
KLEE: done: generated tests = 5
```

These 5 test case is able to achieve high coverage:

```
klee-replay-tests sym.out
...

gcov -b bitonic.cc
...

File 'bitonic.cpp'
Lines executed:100.00% of 30
Branches executed:100.00% of 22
Taken at least once:100.00% of 22
Calls executed:100.00% of 5
```

Another reduction we need to perform is to reduce the number of paths. Currently, you will find that CKLEE runs too long when the number of threads is 8. We will apply some heuristics to cut duplicate branches/paths.