CLSmith tool description

February 13, 2015

1 Introduction

CLSmith is a tool for generating OpenCL C programs. It builds upon Csmith, while keeping Csmith as intact as possible. Programs generated by this tool conform to the OpenCL 1.0 specification.

2 Building CLSmith

The CLSmith repository can be found at `https://github.com/ChrisLidbury/CLSmith`. Use 'git clone' to copy this into a local directory.

Build the generator and launcher as follows:

- In CLSmith root, run '`./configure`'
- In CLSmith root, run '`make`' (it should link fail)
- In [CLSmith-root]/src/CLSmith, run '`make`'

The launcher depends on the OpenCL headers and an implementation of the OpenCL library (libOpenCL.so).

3 Setting up a test directory

This step requires the generator and launcher to have been built successfully. To begin testing, set up a test directory by `cd`ing into [CLSmith-root]/scripts and running '`python cl_setup_test.py path/to/dir`'. This will create a directory at `/homes/$(USER)/path/to/dir/` and populate it with the files necessary for testing.

4 Running experiments

To run experiments, you need to have set up a testing directory and 'cd' into it.

You can run individual tests by invoking the generator yourself and then launching it using 'cl_launcher'. Each generated program will run in a random number of work-items. When each work-item has finished, it will hash its private variables and return it to the launcher, which will then print the hash for all work-items to the standard output. The choice of flags used for the generator will determine what we expect for the printed hash values (see §5 for details).

Run the generated program on a device as follows:

```
./cl_launcher -f <filename> -p <platform-idx> -d <device-idx>
[---debug]
```

or

```
./cl_launcher -f <filename> -n <device_name_contains> [---debug]
```

This will require you to know the platform and device ids on your system.

We provide a script for automating testing:

```
'python cl_get_and_test.py -cl_platform_idx <platform-idx>
-cl_device_idx <device_idx> -zipfile <zip_of_programs>'
```

which will take a zip file of pre-generated programs and run them on the given platform and device. This allows you to generate the programs once, and then run them on multiple configurations. The script will produce a csv file of results.

5 Using the generator

The generator has many features for generating interesting OpenCL programs. The generator is used as follows:

```
./CLSmith [--seed <seed>] [flags]
```

The flags are used to turn on specific features, they are as follows:

- '`--atomic_reductions`': Generate sections of code that will have each thread perform an atomic operation on an element in a buffer with a local variable.
- '`--atomics`': Generate blocks of code guarded by an atomic expression such that threads will enter the block deterministically at the work group level.
- '`--barriers`': Generate barriers throughout the program. Each barrier is surrounded a read from one buffer and a write to a different element in the same buffer, ensuring that any dodgy reordering by the compiler will be shown.
- '`--divergence`': Cause the control flow in the program to diverge by calling '`get_linear_global_id()`' at random points.
- '`--fake_divergence`': Produce expressions that will appear to diverge, but will dynamically evaluate to the same result.
- '`--group_divergence`': Produce random calls to '`get_linear_group_id()`' that will cause threads of different work groups to diverge.
- '`--inter_thread_comm`': Create sections of inter thread communication. At random points in the program, threads in a work group will shuffle their tids (with a barrier just before the shuffle). The buffer is used throughout the program in other expressions.
- '`--vectors`': Use vector types. This include expressions that produce vectors and built in integer functions that work on vectors.

Some combinations of flags are invalid (e.g. `--barriers` and `--divergence`).

Usually, each work-item will produce the same hash, as the programs are entirely deterministic and do not use expressions that evaluate to a different value for different threads. Some features will cause a different hash to be produced by introducing said expressions. These are `--atomic_reductions`, `--atomics`, `--barriers`, `--divergence`, `--group_divergence` and `--inter_thread_comm`. When using any of these flags, you must use 'cl_test_div.py' (shown in §4), otherwise, you should use 'cl_test.py'. All programs are deterministic, and should produce the same set of hash values on each run regardless of platform or device.

6 Design

6.1 AST

Csmith generates a program using ASTs, with the nodes being represented as 'Statement's. There are multiple different statements, with each one encapsulating some behaviour (e.g. 'if' statement and 'for' statement, which will output as an 'if' condition and 'for' loop respectively). Each function is an AST, which is generated post-order using a probability table at each decision point.

In CLSmith, we add new behaviour to the programs by creating a variety of different statements. These are as follows:

- 'AtomicReduction' (in [CLSmith-root]/src/CLSmith/StatementAtomicReduction.{h,cpp}): Outputs as an expression that has each thread perform an atomic operation on a single local variable. If atomics are handled properly, each threads operation should be performed atomically, and the result should be deterministic.

- 'Atomic' (in [CLSmith-root]/src/CLSmith/StatementAtomic.{h,cpp}): Outputs as a block of code guarded by an atomic expression. Only one thread should enter the block, provided atomics are handled properly.

- 'Barrier' (in [CLSmith-root]/src/CLSmith/StatementBarrier.{h,cpp}): Outputs as '`barrier(CLK_LOCAL_MEM_FENCE|CLK_GLOBAL_MEM_FENCE);`'. Optionally, it will also output with some shared buffer code, to help show if the compiler is mishandling the barrier.

- 'Comm' (in [CLSmith-root]/src/CLSmith/StatementComm.{h,cpp}): Each program is given a local buffer, and each thread a tid initialised to its local thread id. The 'comm' node will output an expression that permutes the tids. The buffer is accessed throughout the program using the tid, causing each thread to access different parts of the shared buffer throughout the program.

- 'EMI' (in [CLSmith-root]/src/CLSmith/StatementEMI.{h,cpp}): Outputs as an EMI block, which will be pruned once the AST has been generated. See §6.3 for details.

6.2 Expressions

Expressions work in the same way as the statements. Each expression is represented as an expression tree, that is generated post-order. The difference here, is that expression generation also takes a return type as a parameter, and produces an expression that will return the specified type.

In CLSmith, we add the following expression types:

- 'Atomic' (in [CLSmith-root]/src/CLSmith/ExpressionAtomic.{h,cpp}): Outputs as an atomic expression (e.g. 'atomic_inc(&x)').

- 'ID' (in [CLSmith-root]/src/CLSmith/ExpressionID.{h,cpp}): Outputs as a call to get_global_id(dim), get_local_id(dim), etc.

- 'Vector' (in [CLSmith-root]/src/CLSmith/ExpressionVector.{h,cpp}): Outputs with an intermediate vector (created form sub-expressions) that will then be turned into the expected return type.

We also make use of OpenCL's built-in integer functions (in [CLSmith-root]/src/CLSmith/FunctionInvocationBuiltIn.{h,cpp}). Similarly, this will take the expected return type, and select one of the functions capable of returning that type. It will then convert the type to give the types of the parameters and then generate them.

6.3 EMI Blocks

EMI blocs are created at random points in the program and appear like an if statement. The test expression guarding the block however always dynamically evaluates to `false`. As the code in the block is never executed, the hash generated by the program should not change when the contents of the block is modified. Once the AST has been generated, each EMI block is pruned according to three user supplied probabilities:

- p_{leaf}: Probability of deleting a non-compound statement (a leaf node in the AST).

- $p_{compound}$: Probability of deleting a compound statement, which will in turn delete all of the child nodes in the AST.

- p_{lift}: Probability of deleting a compound statement, and lifting its child nodes to become children of the deleted nodes parents.

The generator is run with the same seed multiple times, but with a different set of probabilities each time. The resulting programs should all give the same hash, despite being different.

7 Testing Modes

The following explains the modes used for testing in the fuzzing paper. All modes include the '`--fake_divergence`' and '`--group_divergence`' flags (see §5 for a description of each flag). Section references refer to the place in the paper where they are described:

- BASIC mode [§4.1]: Simple programs that do not use any of the features provided by OpenCL.

- VECTOR mode [§4.1]: Uses '`--vectors`'.
- BARRIER mode [§4.2]: Uses '`--vectors --inter_thread_comm`'.
- ATOMIC SECTION mode [§4.2]: Uses '`--vectors --atomics`'.
- ATOMIC REDUCTION mode [§4.2]: Uses '`--vectors --atomic_reduction`'.
- ALL mode: Uses '`--vectors --inter_thread_comm --atomics --atomic_reduction`'.