Verification of GPU Programs: Evaluation Challenges

Hannah Zicarelli and Tiago Cogumbreiro

Presenter: Hannah Zicarelli

April 3, 2022

PLACES

Today's talk

Motivation

Our evaluation framework

Conclusion

Motivation

Static analysis of GPU programs

What is this talk about?

- The previous talk (Liew et al.) presented static analysis of GPU data-races
- In this talk we discuss the challenge of large comparative studies in this area
- We focus on NVidia's CUDA: C++ programs that run on GPU hardware
- In this talk we will refer to GPU programs as kernels

Are there any special requirements for static analysis of kernels?

- Static analysis of kernels requires source code annotations

What verifiers are we comparing?

- Static analysis: Faial (our tool), GPUVerify, PUG
- Also, symbolic execution: GKLEE, SESA

Source annotations: static analysis

UMASS BOSTON

Original program source:

```
__global__
void saxpy(int n, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}
```

Source annotations for PUG:

```
#include "my_cutil.h"
assume(blockDim.x == 16);
assume(blockDim.y == 16);
assume(gridDim.x == 64);
assume(gridDim.y == 64);

__global__
void saxpy_kernel(int n, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i]; }
```

- PUG requires that a special C header (`my_cutil.h`) is included
- Static analysis requires problem size (e.g., number of threads) given as assumptions
  - This is the case as SMT solvers cannot handle multiplication of symbolic variables
- PUG expects file extension to be `.c` rather than the normal `.cu`
- PUG requires kernel function to end with string `kernel`

Source annotations: static analysis

```
$ gpuverify --no-inline --only-intra-group --blockDim=16 --gridDim=64
                         --no-benign-tolerance saxpy-gpuverify.cu
```

- GPUVerify can accept the number of threads via command-line arguments
- When employing such command-line arguments, GPUVerify can verify some trivial kernels without additional source annotations
  - This is only the case the only annotation needed is the number of threads
  - Next, we will discuss complex kernels requiring additional annotation
- We implemented our verification tool (Faial) to use GPUVerify style annotations

Source annotations: static analysis

Source annotations for GPUVerify:

```
__global__ void kernel (float* odata, float* idata, int width,
        int height, int nreps) {

    __requires(width == 1024);
    __requires(height == 1024);
    __requires(width > gridDim.y);
    __requires(height > gridDim.x);

    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
    int index  = xIndex + width*yIndex;

    for (int r=0; r < nreps; r++) {
        for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
            odata[index+i*width] = idata[index+i*width];
        }
    }
}
```

Source annotations for PUG:

```
#include "my_cutil.h"

__global__ void kernel (float* odata, float* idata, int width,
        int height, int nreps) {

    assume(width == 1024);
    assume(height == 1024);
    assume(width > gridDim.y);
    assume(height > gridDim.x);

    assume(blockDim.x == 16);
    assume(blockDim.y == 16);
    assume(gridDim.x == 64);
    assume(gridDim.y == 64);

    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
    int index  = xIndex + width*yIndex;

    for (int r=0; r < nreps; r++) {
        for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
            odata[index+i*width] = idata[index+i*width]; } } }
```

- Additional code annotations are often needed to constrain analysis
- These must be source annotations, and annotation syntax differs by verification tool

Source annotations: static analysis

UMASS BOSTON

Annotation	*GPUVerify*	*PUG*
Preconditions	`__requires(...);`	`assume(...);`
File extension	`.cu`	`.c`
Number of threads	command-line or source	source annotation
Required headers		`"my_cutil.h"`

Additionally, PUG places further restrictions on the C++ kernel source code:

- No C++ templates
- No classes
- All loops must be `for` loops and face additional restrictions

GPUVerify and PUG require differing source annotations!

Source annotations: symbolic execution

UMASS BOSTON

Source annotations for GKLEE:

```
__global__
void saxpy(int n, float a, float *x, float *y) {

    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i]; }

int main () {
     int n;
     klee_make_symbolic(&n, sizeof(int), "n");
     float a;
     klee_make_symbolic(&a, sizeof(float), "a");
     float *x;
     cudaMalloc((void**)&x, 1024 * sizeof(float));
     float *y;
     cudaMalloc((void**)&y, 1024 * sizeof(float));
     dim3 grid_dim(16);
     dim3 block_dim(64);
     saxpy<<< grid_dim, block_dim >>>(n, a, x, y);
     return 0;
}
```

Symbolic execution annotations:

- A `main` function is required to be an execution entry point
- Each kernel parameter must be initialized (symbolically)
- The problem size (e.g., number of threads) must be specified
- Finally, the main function must invoke the kernel function
- Note that this main function specific to symbolic evaluation
    - This main function won't work for running the kernel on a GPU

Related work

- Studies for the following verifiers compare two or fewer total verifiers:
  - 2 for GPUVerify (2012 by Betts et al.)
  - 1 for PUG (2012 by Li and Gopalakrishnan)
  - 2 for GKLEE (2012 by Guodong et al.)
  - 2 for SESA (2014 by Li, Li, and Gopalakrishnan)
- Studies for the following verifiers have limited average lines of code (LoC) analyzed:
  - 13 avg. LoC for ESBMC-GPU (2016 by Pereira et al.)
  - 16 avg. LoC for Simulee (2020 by Wu et al.)

We survived existing published studies: they lack tool diversity and depth!

The challenge of large evaluations

- Each verifier requires tool-spesific code annotations
- These code annotations are incompatable across verifiers
- Dataset must maintain variations of each program for each verifier

This maintance would be highly labor intensive for researchers!

Source annotations are an impedement to large evaluations

Our evaluation framework

A tool-agnostic kernel format

```
grid_dim = [ 64 ]
block_dim = [ 16 ]

pass = true

body = '''
  int i = blockIdx.x*blockDim.x
          + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
'''

scalars = [{n = "int"}, {a = "float"}]
arrays = [{x = "float"}, {y = "float"}]
```

Our tool-agnostic format can specify:

- Number of threads (`block_dim`, `grid_dim`)
- C headers to be included
- Kernel function parameters, by type
- Wether the test passes (e.g., data-race freedom, barrier divergence, etc)
- Preamble code
- Preconditions (e.g., invariants)
- The body of the kernel function

- We use TOML (Tom's Obvious Minimal Language) to structure this data
- Why TOML? TOML libraries exist to be used with most programming languages

Automation

We have built an ecosystem of tools on top of this tool-agnostic kernel format. Examples include:

- Verifier-specific kernel generation
- Running multiple verifiers on a kernel
- Gathering metrics on code features for each kernel
- Conversion from CUDA to tool-agnostic kernel format

We will cover each of these tasks in the following slides

Verifier-specific kernel generation

Snippet from the GPUVerify template that generates preconditions:

```
{% if pre | length > 0 %}
  /* kernel pre-conditions */
  {% for p in pre %}
    __requires({{ p }});
  {% endfor %}
{% endif %}
```

- We employ the Jinja template engine and Python to generate programs
- Jinja (programming language agnostic) is commonly used to generate HTML web pages

We developed a tool `kernel-gen.py` to generate kernels:

- Input is the tool-agnostic format
- A command-line argument specifies the format of the output program
- Output is a CUDA kernel formatted for the specified verifier

Running multiple verifiers on a kernel

```
$ test-tools.py saxpy.toml
RUN timeout 60 faial --parse-gv-args saxpy-faial.cu
RUN timeout 60 gpuverify --no-inline --only-intra-group --blockDim=16 --gridDim=64
                         --no-benign-tolerance saxpy-gpuverify.cu
RUN timeout 60 pug saxpy-pug.c
  status    time    memory  data_races    tool
--------  ------  --------  ------------  ---------
       0    0.18    48.418  drf           faial
       0    1.34   37.5859  drf           gpuverify
       0    0.05   50.6484  drf           pug
```

- Some verifiers require metadata stored in the tool-agnostic kernel format
- For example, recall that GPUVerify needs the number of threads

Gathering metrics on code features

We extend our tool-agnostic kernel format to store metadata on code features of the dataset. The following is an example of such metadata:

```
max_sync_nesting = 1
sync_loop_count = 1
unsync_loop_count = 1
loop_count = 2
write_count = 3
read_count = 4
if_count = 0
sync_count = 2
line_count = 71
```

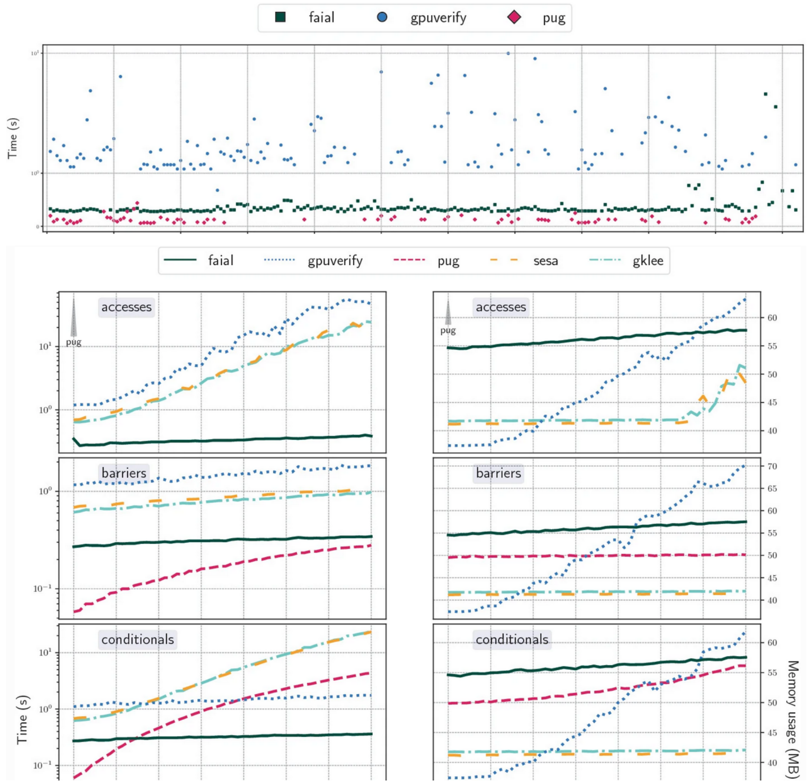
This is possible as the TOML format is easily extensible

Conversion from CUDA to agnostic format

- Much of our evaluation is based on on GPUVerify's CAV 2014 dataset
- This dataset of 227 CUDA kernels comes with GPUVerify source annotations
- We employed a script to convert this CUDA dataset to our tool-agnostic format
- The script (~150 lines of Python) consists of rudimentary text processing
- This conversion was mostly-automated, handling 65% of the dataset

Takeaway: a little automation saves a lot of time!

Results: CAV 2021 evaluation



This benchmarking infrastructure enabled multiple experiments:

- A dataset of 227 CUDA kernels, 3 static verifier, including kernels with up to 850 lines of code
- And an additional 250 synthetic kernels, 5 static and symbolic verifiers
- To our knowledge, this is the largest published experimental evaluation of GPU verifiers to date

Conclusion

Future challenges

Support for numeric and symbolic problems

- Currently, we check verifier output for boolean correctness tests
- In the future, we are interested in parsing output for numeric and symbolic output

Weakening and strengthening preconditions

- We are interested in which preconditions affect certain properties

What we learned

- Source annotations are an impediment to large evaluations
- Templates aid the handling of differing annotations via program generation
- A little automation goes a long way

With templates and automation in place, larger evaluations are feasible.