More blogs in Graphics and Gaming

Graphics and Gaming blog

Tell us what you think

Tags

OpenCL

Mali

gpu_compute

mali_graphics_debugger

4

Actions

Feedback

Debugging OpenCL Applications with Mali Graphics Debugger V2.1 and GPUVerify

Stephen Barton April 14, 2015

Most people use Mali Graphics Debugger (MGD) to help debug OpenGL® ES applications on Linux or Android. However graphics is not the only API that is supported by MGD, in-fact MGD supports applications that use the OpenCL™ API as well. This means that if you run MGD with an application that uses OpenCL you will get the same function level tracing that you shader source code.

With the release 2.1 of Mali Graphics Debugger the OpenCL feature set has been improved by the inclusion of GPUVerify support. GPUVerify is a tool for formal analysis of kernels written in OpenCL and was partly funded by the EU FP7 CARP project http://carpproject.eu/. The tool can prove that kernels don’t suffer from the following three issues:

- Intra-group data races: This is when there is a data race between work items in the same work group.
- Inter-group data races: This is when there is a data race between work items in different work groups.
- Barrier divergence: This is when a kernel breaks the rules for barrier synchronization in conditional code defined in the OpenCL documentation.

The tool was created by Imperial College London and more information about the tool and the issues it can diagnose can be found by visiting http://multicore.doc.ic.ac.uk/GPUVerify.

In essence if you provide GPUVerify with the source code of a OpenCL kernel along with the local and global work group sizes it can check your

Related blog posts

Live editing OpenGL ES shaders with Mali Graphics Debugger
With Mali Graphics Debugger you can edit Open...

Building a Unity Application with Mali Graphics Debugger Support
Update: An updated version of how to build Unity...

What's new in Mali Graphics Debugger 2.1 and OpenGL ES Emulator 2.1
Every year at GDC, we like to present some impor...

Mali OpenCL Flag Demo
This is a demo created internally at ARM by Anth...

Building an Unreal Engine Application with Mali Graphics Debugger Support
In a previous blog we talked about running Mali G...

Related forum threads

Mali Graphics Debugger - debugging native Android application
Is it possible to debug native Android application ...

Making Mali Graphics Debugger work with opencl on Odroid Xu3
Hey! I'm trying to debug an opencl application on ...

Feedback

Mali OpenCL SDK compile problem
Hello, I am trying to run the 'hello_world_opencl' ...

Mali OpenCL SDK Linux Cross-Compiler
Hello to all, I am using a Debian Linux host compu...

run GPUVerify is already known from tracing your application. The steps needed to use this tool with MGD are outlined below:

Step 1: Download Mali Graphics Debugger v2.1 and GPUVerify

Mali Graphics Debugger can be downloaded from http://malideveloper.arm.com/develop-for-mali/tools/software-tools/mali-graphics-debugger/. If you are using an older version of MGD you will need to upgrade. Version 2.1 has much more included than just GPUVerify support, a few items are listed below:

- Support for ARMv8 (Android 64-bit)
- Support for tracing Android Extension Pack
- Many improvements to the Frame Overrides feature.

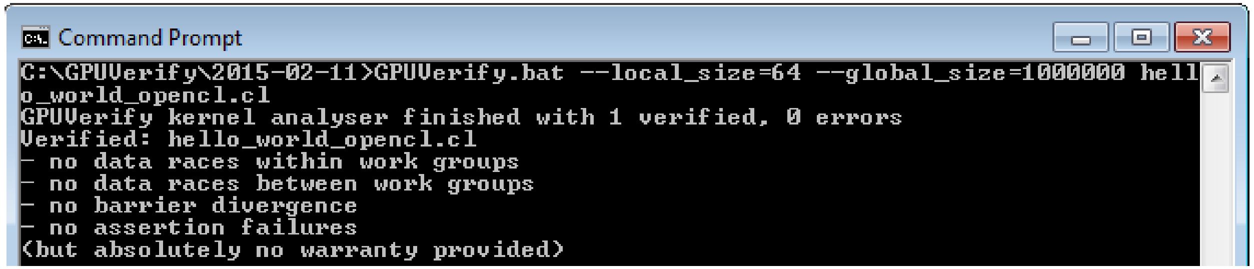
GPUVerify can be downloaded in binary form for both Linux and Windows from the following location: http://multicore.doc.ic.ac.uk/tools/GPUVerify/download.php. GPUVerify does support Mac OS X as well but you will have to build it from source.

Feedback

stand-alone

Using GPUVerify successfully inside MGD is often easier once it is established that GPUVerify works in a stand-alone capacity. Once it has been downloaded the process of making it work is easy thanks to the documentation provided on the GPUVerify's website. Along with a section that provides common troubleshooting advice. A good way to check that it is working correctly is to try GPUVerify with some of the examples that are provided with the Mali OpenCL SDK.

The following image is of the output from GPUVerify running the HelloWorld example from the Mali OpenCL SDK:

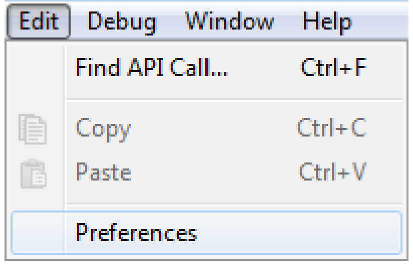


```
C:\GPUVerify\2015-02-11>GPUVerify.bat --local_size=64 --global_size=1000000 hell
o_world_opencl.cl
GPUVerify kernel analyser finished with 1 verified, 0 errors
Verified: hello_world_opencl.cl
- no data races within work groups
- no data races between work groups
- no barrier divergence
- no assertion failures
(but absolutely no warranty provided)
```

Location of GPUVerify

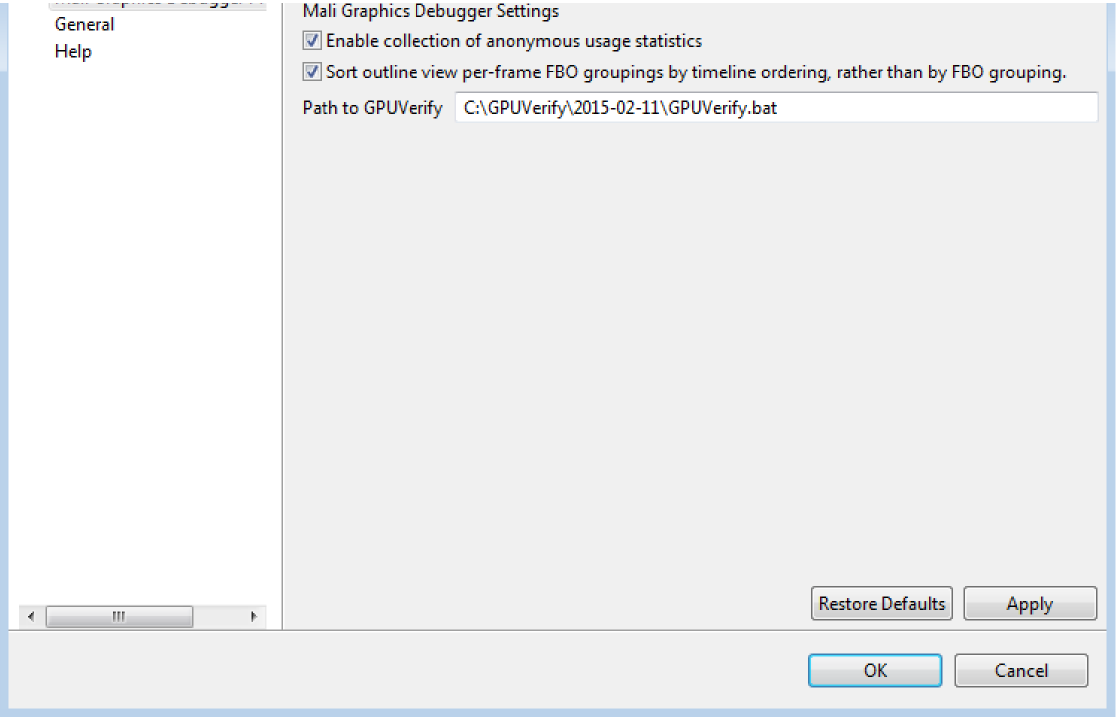
As GPUVerify is not shipped with MGD you must specify the location of where you installed it to MGD. To do this in MGD do the following:

- Click Edit -> Preferences



- Click on the Text box next to "Path to GPUVerify" and provide the location of the GPUVerify binary.

Feedback



Mali Graphics Debugger Settings

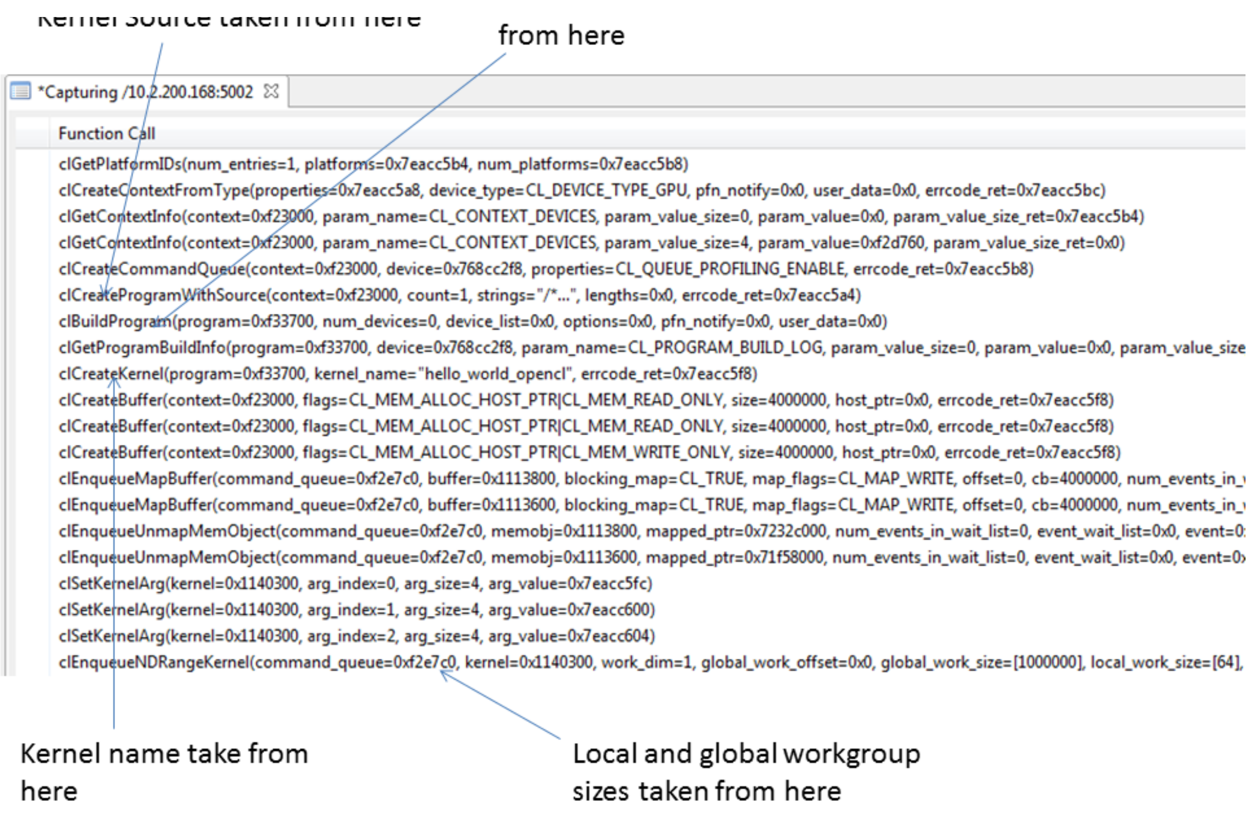
☑ Enable collection of anonymous usage statistics

☑ Sort outline view per-frame FBO groupings by timeline ordering, rather than by FBO grouping.

Path to GPUVerify C:\GPUVerify\2015-02-11\GPUVerify.bat

Step 4: Run a normal OpenCL trace in MGD

As mentioned previously MGD will try to fill in the prerequisite information for GPUVerify from the trace. It does this by looking for several functions; the most important are `clCreateProgramWithSource`, `clCreateKernel` and `clEnqueueMapBuffer`.


Kernel Source taken from here
from here
Kernel name take from here
Local and global workgroup sizes taken from here

```
*Capturing /10.2.200.168:5002

Function Call
clGetPlatformIDs(num_entries=1, platforms=0x7eacc5b4, num_platforms=0x7eacc5b8)
clCreateContextFromType(properties=0x7eacc5a8, device_type=CL_DEVICE_TYPE_GPU, pfn_notify=0x0, user_data=0x0, errcode_ret=0x7eacc5bc)
clGetContextInfo(context=0xf23000, param_name=CL_CONTEXT_DEVICES, param_value_size=0, param_value=0x0, param_value_size_ret=0x7eacc5b4)
clGetContextInfo(context=0xf23000, param_name=CL_CONTEXT_DEVICES, param_value_size=4, param_value=0xf2d760, param_value_size_ret=0x0)
clCreateCommandQueue(context=0xf23000, device=0x768cc2f8, properties=CL_QUEUE_PROFILING_ENABLE, errcode_ret=0x7eacc5b8)
clCreateProgramWithSource(context=0xf23000, count=1, strings="/*...", lengths=0x0, errcode_ret=0x7eacc5a4)
clBuildProgram(program=0xf33700, num_devices=0, device_list=0x0, options=0x0, pfn_notify=0x0, user_data=0x0)
clGetProgramBuildInfo(program=0xf33700, device=0x768cc2f8, param_name=CL_PROGRAM_BUILD_LOG, param_value_size=0, param_value=0x0, param_value_size
clCreateKernel(program=0xf33700, kernel_name="hello_world_opencl", errcode_ret=0x7eacc5f8)
clCreateBuffer(context=0xf23000, flags=CL_MEM_ALLOC_HOST_PTR|CL_MEM_READ_ONLY, size=4000000, host_ptr=0x0, errcode_ret=0x7eacc5f8)
clCreateBuffer(context=0xf23000, flags=CL_MEM_ALLOC_HOST_PTR|CL_MEM_READ_ONLY, size=4000000, host_ptr=0x0, errcode_ret=0x7eacc5f8)
clCreateBuffer(context=0xf23000, flags=CL_MEM_ALLOC_HOST_PTR|CL_MEM_WRITE_ONLY, size=4000000, host_ptr=0x0, errcode_ret=0x7eacc5f8)
clEnqueueMapBuffer(command_queue=0xf2e7c0, buffer=0x1113800, blocking_map=CL_TRUE, map_flags=CL_MAP_WRITE, offset=0, cb=4000000, num_events_in_
clEnqueueMapBuffer(command_queue=0xf2e7c0, buffer=0x1113600, blocking_map=CL_TRUE, map_flags=CL_MAP_WRITE, offset=0, cb=4000000, num_events_in_
clEnqueueUnmapMemObject(command_queue=0xf2e7c0, memobj=0x1113800, mapped_ptr=0x7232c000, num_events_in_wait_list=0, event_wait_list=0x0, event=0
clEnqueueUnmapMemObject(command_queue=0xf2e7c0, memobj=0x1113600, mapped_ptr=0x71f58000, num_events_in_wait_list=0, event_wait_list=0x0, event=0x
clSetKernelArg(kernel=0x1140300, arg_index=0, arg_size=4, arg_value=0x7eacc5fc)
clSetKernelArg(kernel=0x1140300, arg_index=1, arg_size=4, arg_value=0x7eacc600)
clSetKernelArg(kernel=0x1140300, arg_index=2, arg_size=4, arg_value=0x7eacc604)
clEnqueueNDRangeKernel(command_queue=0xf2e7c0, kernel=0x1140300, work_dim=1, global_work_offset=0x0, global_work_size=[1000000], local_work_size=[64],
```

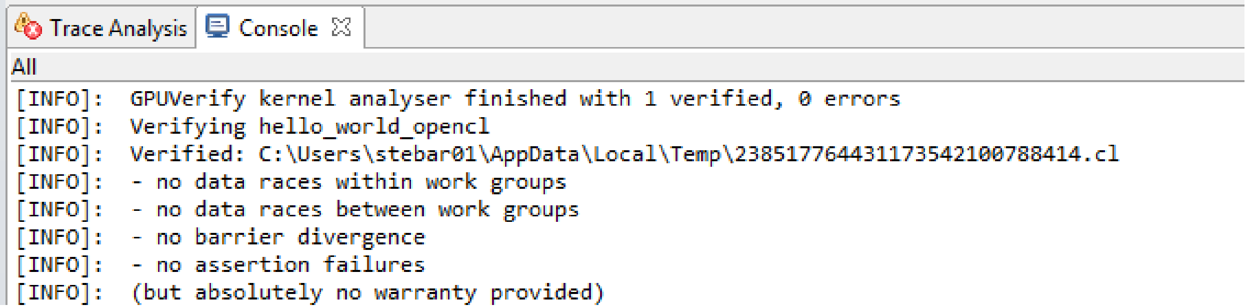
If you don't use `clCreateKernel` to create your kernels, MGD can also obtain the information from `clCreateKernelsInProgram`. As shown in the image above MGD also captures the build options used in `clBuildProgram` to pass to GPUVerify. The more details that can be passed to GPUVerify the more accurate it can analyze your kernels.

To run the tool you need to select Debug -> Launch GPU Verify. MGD will then present a new dialog box which summarizes all of the information MGD managed to pull out of the trace. You are free to fill this information in or even change the information. One of the reasons you may want to do this is to try new work group sizes or global work sizes, to see if there are any unforeseen issues with different kernel parameters.

Feedback

Stage 6: Analyzing the Results

The results should be placed in the console view as part of MGD. Here are the results of the HelloWorld example from above running through MGD:



```
[INFO]:  GPUVerify kernel analyser finished with 1 verified, 0 errors
[INFO]:  Verifying hello_world_opencl
[INFO]:  Verified: C:\Users\stebar01\AppData\Local\Temp\2385177644311735421007888414.cl
[INFO]:  - no data races within work groups
[INFO]:  - no data races between work groups
[INFO]:  - no barrier divergence
[INFO]:  - no assertion failures
[INFO]:  (but absolutely no warranty provided)
```

Mali Graphics Debugger can be used to do much more than debug and trace graphics applications. It can be used to debug and trace OpenCL applications as well. With the inclusion of GPUVerify support it is now possible to debug possible data race conditions in your kernels as well as barrier divergence issues. MGD will send as much data as it can to GPUVerify by analyzing the trace of your OpenCL application.

Feedback

Leave a comment...

Edit ▾ Insert ▾ Format ▾ Tools ▾ Login and Comment

Oldest Best Newest

● Anton Lokhmotov over 6 years ago

Brilliant! Thank you so much the Mali Tools team!
I must add that it's a culmination of three years of effort in the EU-funded CARP project: carpproject.eu. Watch for other good things to come out soon!