GPU Encrypt – First Progress Report

James Gleeson, Sreekumar Rajan, Vandana Saini

October 23rd

Achievements:

Determine the status of OpenCL access to the Adreno 320 GPU on the Nexus 4 Android platform.

As later sections in our report will reveal, our attempts at running an OpenCL AES implementation and other trivial user-level OpenCL programs on the Adreno 320 GPU of the Nexus 4 phone running an Android 4.2.2 version proved unstable when activating the display of the device in the middle of their execution.

To quickly rule out that this unexpected behaviour was due to us running an outdated Android version, we decided to rerun the tests on the latest 4.3 Android release for the Nexus 4 phone. Our attempts to run the OpenCL AES algorithm resulted in a segmentation fault showing that the OpenCL shared library was missing from its usual location: `/system/lib/libOpenCL.so`. Investigating the proprietary blobs provided by Google [5] reveal that it has been removed as of 4.3. Indeed, OpenCL will no longer be accessible by developers starting with Android 4.3, and the new standard for accessing general-purpose GPU programming (GPGPU) capabilities is through RenderScript [8]. Hence, the subsequent sections of our report investigate OpenCL stability on the Android 4.2.2 platform.

Find an existing user-level AES encryption implementation that makes use of OpenCL

We obtained an existing OpenCL implementation of the AES encryption algorithm from a combination of sources. In particular, we first obtained a starter application illustrating the usage of OpenCL on the Android platform, which integrates with the Native Development Kit (NDK) build tools for compiling native C/C++ compatible with the ARM-based CPU of the Nexus 4 phone [1, 5]. We confirmed the correct usage of the OpenCL API in this starter program by cross-referencing the order of calls for allocating and deallocating OpenCL resources (e.g. kernel context, sending OpenCL kernel parameters, reading OpenCL memory buffers to obtain output etc.) from a tutorial on OpenCL programming [3]. Next, we obtained an existing implementation of the AES algorithm in OpenCL from a separate Github repository [4]. Since OpenCL is a general-purpose programming framework, code that makes use of it does not require device specific knowledge of the available GPU or CPU [7]. Hence, no modifications were needed to the kernel implementation of the AES algorithm.

The OpenCL AES test program (`opencl_aes`) operates by allocating a fixed size array (128 bytes) of zeroed data, and makes use of a fixed 128-bit key to encrypt the zeroed data. This data is then transferred to the GPU device memory via the OpenCL `clEnqueueWriteBuffer` command and the AES kernel is executed using `clEnqueueNDRangeKernel` [7].

When the AES kernel finishes, the `clWaitForEvents` command returns and the output is read from the GPU device via `clEnqueueReadBuffer` [7]. The correct operation of this program is asserted by checking the value returned when reading back encrypted data from the GPU, and ensuring that the result is consistent across runs.

Our first experiment investigated the stability of the OpenCL library when no interaction was being performed with the device. In particular, we ran the `opencl_aes` program 100 consecutive times, and investigated the encrypted data. Table 1 indicates the input data (zeroed data) and the observed encrypted data for all 100 consecutive runs:

Input	Observed Output
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	6A 50 AE 0C 58 8A 89 61 40 A6 70 83 43 E8 F7 91
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	6A 50 AE 0C 58 8A 89 61 40 A6 70 83 43 E8 F7 91
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	6A 50 AE 0C 58 8A 89 61 40 A6 70 83 43 E8 F7 91
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	6A 50 AE 0C 58 8A 89 61 40 A6 70 83 43 E8 F7 91
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	6A 50 AE 0C 58 8A 89 61 40 A6 70 83 43 E8 F7 91
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	6A 50 AE 0C 58 8A 89 61 40 A6 70 83 43 E8 F7 91
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	6A 50 AE 0C 58 8A 89 61 40 A6 70 83 43 E8 F7 91
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	6A 50 AE 0C 58 8A 89 61 40 A6 70 83 43 E8 F7 91

Table 1: 100 consecutive runs of `opencl_aes` without interaction on the phone.

Thus, the output was consistent across all 100 consecutive, indicating that the OpenCL library behaves correctly when there is no contention by other processes to make use of the GPU (e.g. OpenGL or other OpenCL programs).

Next, we wished to test whether the OpenCL library (and underlying driver) implements coordination between other processes that try to use of the GPU concurrently. In particular, we conduct the 100 consecutive run experiment again, but this time we test for interference with Android processes that use OpenGL for hardware acceleration by activating the display of the device in the middle of the execution.

Input	Observed Output
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Table 2: All subsequent consecutive runs of opencl_aes after the display becomes active.

As indicated by Table 2, on the 63rd and all subsequent runs of the `opencl_aes` program (coinciding with activation of the display), the output read from the GPU device memory is all zero data. It was observed that the phone becomes unresponsive in this state (to touch or button presses).

To further evaluate the effects of this misbehaviour, snapshots of process activity were collected before and after the failed execution using the `top` command.

Process Name
`android.process.media`
`com.android.launcher`
`com.android.location.fused`
`com.android.mms`
`com.android.musicfx`
`com.android.nfc`
`com.android.nfc:handover`
`com.android.phasebeam`
`com.android.phone`
`com.android.providers.calendar`
`com.android.systemui`
`com.android.vending`
`com.google.android.apps.maps:LocationFriendService`
`com.google.android.apps.walletnfcrel`
`com.google.android.calendar`
`com.google.android.deskclock`
`com.google.android.gm`
`com.google.android.googlequicksearchbox`
`com.google.android.gsf.login`
`com.google.android.inputmethod.latin`
`com.google.android.music:main`
`com.google.android.onetimeinitializer`
`com.google.android.partnersetup`
`com.google.android.talk`
`com.google.android.youtube`
`com.google.process.gapps`
`com.google.process.location`
`eu.chainfire.supersu`
`flush-0:17`
`kworker/u:0`
`kworker/u:1`
`kworker/u:3`

Table 3: Processes that have terminated post-failure when activating the display during `opencl_aes`

As observed above, most of the processes that have stopped responding are Android user applications (as indicated by the `com.google.*` and `com.android.*` prefixes in their name) [9]. This result indicates that the OpenCL library is not coordinating access to the GPU, and since Android applications are crashing (which typically make use of the GPU for hardware acceleration), OpenGL tasks are probably not being coordinated across processes with by the OpenCL driver.

To rule out missing use of coordination API calls, we investigated the OpenCL Programming Guide [7]. Two methods exist for synchronizing GPU access across OpenCL and OpenGL. In implicit synchronization, any OpenGL command that accesses the contents of an OpenGL memory object listed in the argument list of `clEnqueueAcquireGLObjects()` and were issued prior to the call to `clEnqueueAcquireGLObjects()` will complete before execution of any OpenCL commands following the `clEnqueueAcquireGLObjects()`. In explicit synchronization, an OpenCL event is created from an OpenGL fence sync object via the function `clCreateEventFromGLSyncKHR()`. The OpenGL thread calls `glFenceSync()` to ensure any OpenCL commands have executed prior to executing its OpenGL commands, and the OpenCL thread calls `clEnqueueAcquireGLObjects()` (on the fence sync object event) to ensure OpenGL commands prior to the call have finished. Hence, the explicit synchronization is useful when an OpenGL thread separate from the OpenCL thread is accessing the same underlying memory object.

Unfortunately, a major limitation of both the implicit and explicit synchronization is that they assume OpenGL and OpenCL commands occur in the same process with a shared memory region for storing shared memory objects, whereas we are interested in mutual exclusion of the GPU across different processes that do not share memory (Android processes in Table 3 and our `opencl_aes` test program). Hence, we were unable to use this approach.

We also investigated OpenCL functions that would give us information about the availability of a GPU before using it. `clGetDeviceInfo()` returns information about an OpenCL device [7]. The enumeration constant `CL_DEVICE_AVAILABLE` can be passed as the second argument and the function will return `CL_TRUE` to indicate the availability of a device or `CL_FALSE` otherwise. However, during our experimental runs, we found it was never the case that `CL_FALSE` was returned and instead the OpenCL runtime system always indicated availability of a device. We also performed exhaustive error checking for all OpenCL commands (i.e. ensuring all calls return with `CL_SUCCESS`) to the OpenCL implementation of AES encryption algorithm. However, `CL_SUCCESS` was always returned, even in cases when the test program would fail.

Create user-level programs for interacting with the Adreno 320 GPU through OpenCL to determine if it is stable.

To further rule out the possibility that the observed behaviour could be due to improper usage of the OpenCL library (e.g. unchecked error conditions, missing use of coordination API calls) or bugs in the provided AES code, a minimal OpenCL program named helloworld was constructed that writes 1KB of `'1'` characters and uses one GPU to increment them to `'2'` characters.

On running this program with the display off, everything works as expected: 1KB of `'1'`s are read into GPU device memory and 1KB of `'2'`s are read out of GPU device memory after completion of the trivial OpenCL kernel.

However, if the display of the phone is activated simultaneously with the OpenCL program, the same behaviour as was described for the `opencl_aes` program is observed in the `helloworld` program. In particular, the output read from the GPU device memory is 1KB of zeroed data.

The further investigate what level of coordination is implemented by the GPU, we investigated running the `helloworld` program with up to 1000 concurrently executing instances, without activating the display. Thus, here we are testing whether the OpenCL library is implementing coordination amongst concurrently executing OpenCL kernels (but not any interaction with OpenGL).

Input/Output Buffer	Value
Input	1111111111...
Output	2222222222...
Output	2222222222...
Input	1111111111...
Output	2222222222...

Table 4: Snapshot of concurrently executing and coordinated OpenCL (`helloworld`) programs

As indicated in table 4, the interleaved printing of input and output buffers of the concurrently executing `helloworld` processes without any output arrays resulting in zeroed data is evidence that the OpenCL library is providing coordination amongst the execution of OpenCL kernels belonging to separate processes (in contrast to OpenCL and OpenGL).

Consult experts in the field about past experience with GPGPU programming.
With limited success in coordinating GPU access on the Nexus 4 phone, we resorted to consulting two technical experts in the field of GPGPU programming. The first expert is a PhD student named Sahil Suneja who has had experience in OpenCL programming on desktop platforms with discrete graphics cards as well as laptops with integrated graphics chips. Sahil indicated that he runs a typical GUI environment that makes use of the GPU concurrently with OpenCL programs, and has indicated no previous experience similar to the problems we are experiencing.

We had an opportunity to speak with a technical representative from QualComm on campus (reachable at farid2@qti.qualcomm.com), who happened to be a driver developer, about the status of coordinated access to the GPU. QualComm manufactures the SnapDragon chipset found in the Nexus 4 phone, and is responsible for distributing the proprietary OpenCL drivers to Android platforms. We indicated the instability issues we were facing with the coordination of OpenCL and OpenGL accessing the GPU. He suggested implementing solutions involving shared memory across the processes using a shared frame buffer. This would however require coordination of GPU access via a single “master” process (requiring instrumentation of all existing Android processes that make use of OpenGL to instead communicate with the “master” process). The representative further confirmed that there is no coordination amongst OpenGL and OpenCL as implemented in the OpenCL driver, which would explain our experimental observations.

Achievements for the next milestone (November 13th):

Expert comments and our in-depth study indicate that we must further investigate the mechanisms by which GPU coordination (amongst OpenCL and OpenGL) is implemented in mature desktop machines where our observed unexpected behaviour is absent. Once we have determined how such activity is coordinated, we intend to investigate the practicality of implementing this behaviour on the Android platform. The technical barriers to accomplishing this are that the OpenCL drivers are proprietary (provided by QualComm), so if the coordination must be implemented there, we will most likely be unable to proceed.

Benchmark the performance of both algorithms on the Nexus 4 for battery consumption (amplitude during execution) and throughput (rate of encryption).
Due to unexpected difficulties in assessing the stability of the OpenCL library on the Android platform, this task has been postponed until future milestones. Indeed, the future milestones are also subject to change as this project progresses.

References:

[1] aopencl, *Android OpenCL examples*, http://code.google.com/p/aopencl/
[2] Android Developers, *Android NDK*, http://developer.android.com/tools/sdk/ndk/index.html
[3] CodePlex, *OpenCL*, http://opencl.codeplex.com/wikipage?title=OpenCL%20Tutorials%20-%201
[4] Github, *OpenCL-AES*, https://github.com/softboysxp/OpenCL-AES
[5] Google Developers, *Binaries for Nexus Devices* – https://developers.google.com/android/nexus/drivers#makojwr66y
[6] Google Inc., *Nexus 4*, http://www.google.com/nexus/4/.
[7] Munshi, Aaftab, et al. *OpenCL programming guide*. Pearson Education, 2012.
[8] OpenCL Blog, *OpenCL Disabled on Android* – http://www.openclblog.com/2013/08/opencl-disabled-on-android.html?m=1
[9] Yaghmour, Karim. *Embedded Android: Porting, Extending, and Customizing*. O'Reilly, 2013.