GPU Encrypt – Second Progress Report

James Gleeson, Sreekumar Rajan, Vandana Saini
November 13

Achievements:

In the past report, we discussed our experiments investigating the stability of OpenCL on the Nexus 4 Android platform. This report will discuss our futher efforts looking at stability and our attempts at gathering benchmark results for the OpenCL kernel on the Nexus 4. Next, we will discuss our transition to the MotoX phone which offers a more stable OpenCL implementation, and our results thus far for benchmarking GPU AES encryption.

Benchmark the performance of OpenCL AES encryption on the Nexus 4 for throughput (rate of encryption).

Methodology

Before benchmarking the OpenCL AES encryption algorithm, we investigated prior studies that looked at benchmarking GPU implementations of algorithms. In particular, Che et al. [1] note that one of the common sources of inefficiency when going from a CPU to a GPU implementation of an algorithm is caused by the disjoint address spaces of the CPU and GPU, and hence a need transfer memory between them on startup of an OpenCL kernel (CPU-to-GPU) and finishing of an OpenCL kernel (GPU-to-CPU). In addition to the cost in copy overhead, there is also the cost associated with interacting with and activating the GPU driver (which we refer to as startup cost).

To collect timing measurements, we make use of the OpenCL API which defines a function clGetEventProfilingInfo for obtaining profiling information for an event in an OpenCL program. Events in OpenCL programs might correspond to a GPU-to-CPU memory copy (in the case of corresponding to a clEnqueueWriteBuffer call), a CPU-to-GPU memory copy (a clEnqueueReadBuffer call), or the OpenCL kernel execution (a clEnqueueNDRangeKernel call).

When commands are added to the command buffer, they return an event identifier. Using this event identifier, the OpenCL clGetEventProfilingInfo command allows one to get profiling information, in particular, the start (CL_PROFILING_COMMAND_START) and end (CL_PROFILING_COMMAND_END) times for the corresponding command.

Using this API, we aimed to gather three measurements

1. **Inherent overestimate:** The time it takes an empty OpenCL kernel to run (i.e. only the OpenCL kernel's runtime, not including CPU-GPU communication or startup time). This measurement allows us to assess whether our OpenCL kernel execution measurements returned by the API have any inherent overestimations that we should account for.
2. **CPU-to-GPU:** The time it takes to copy an input array from main memory to the GPU over varying input sizes.
3. **GPU-to-CPU:** The time it takes to copy an output array from the GPU back to main memory varying output sizes.
4. **AES encryption:** Measure the time it takes to run the OpenCL AES kernel on different sizes of input, including only the kernel runtime and not time for GPU-CPU communication.

Results

Inherent overstimate

To measuring the inherent overstimate, 25 consecutive measurements were performed. The OpenCL kernel simply calls the get_global_id() function then exits (to ensure the OpenCL runtime does not perform any optimization that prevents the kernel from running at all).

Minimum	First Quartile	Median	Mean	Third Quartile	Maximum
1e-06 ms	1e-06 ms	1e-06 ms	1e-06 ms	1e-06 ms	1e-06 ms

Table 1: Summary statistics for inherent overestimates in OpenCL kernel execution.

From table 1, it's clear that the OpenCL API does not overestimate the time for an OpenCL kernel to execute (all measurements were 1 nanosecond), so we need not correct for any overestimations when we later measure AES encryption times.

CPU-to-GPU and GPU-to-CPU

To measure CPU-GPU communication, we collected 10 consecutive measurements ranging from zero up to 512MB in constant increments (i.e. Of 51.2MB). Table 2 below shows the results for CPU-to-GPU communication. The GPU-to-CPU communcation results are not shown since they are identical to table 2.

Minimum	First Quartile	Median	Mean	Third Quartile	Maximum
1e-06 ms	1e-06 ms	1e-06 ms	1e-06 ms	1e-06 ms	1e-06 ms

Table 2: Summary statistics for CPU-to-GPU communication.

From table 2 we surprisingly observe that the profiling times returned by the OpenCL API are negligible (again 1 nanosecond).

To rule out that we are calling the OpenCL functions improperly (e.g. collecting profiling information prior to an operation finishing) we further investigated the OpenCL API for the flags that are passed to clEnqueueWriteBuffer and clEnqueueReadBuffer functions. These flags offer options for performing either a blocking write (or read), or to perform it asynchronously and synchronize via a call to clFinish prior to gathering any profiling information from clGetEventProfilingInfo. The specification for clCreateBuffer has a flag (CL_MEM_USE_HOST_PTR) for a CPU-to-GPU copy that permits "caching" of input data in GPU memory, implying that it may do a copy to GPU memory on demand. Hence, it is not clear whether this behaviour would result in the copy being factored into the OpenCL kernel execution time. To prevent this unexecpted overestimate of OpenCL kernel execution time, we ensure in our memory CPU-to-GPU copying has been performed prior to executing an OpenCL kernel.

In summary, it appears that the OpenCL API is not providing reliable profiling information in these measurements. If we wish to gather more accurate CPU-GPU timing information, we will consider using blocking calls intermixed with calls to timers external to the OpenCL API.

AES encryption on the Nexus 4:
To measure the rate of encryption, we collected measurements over varying input sizes starting with 128 bytes and increasing by powers of 2 up to 8K. That is, we were only able to collect 7 measurements (this limitation will be discussed in more detail below).

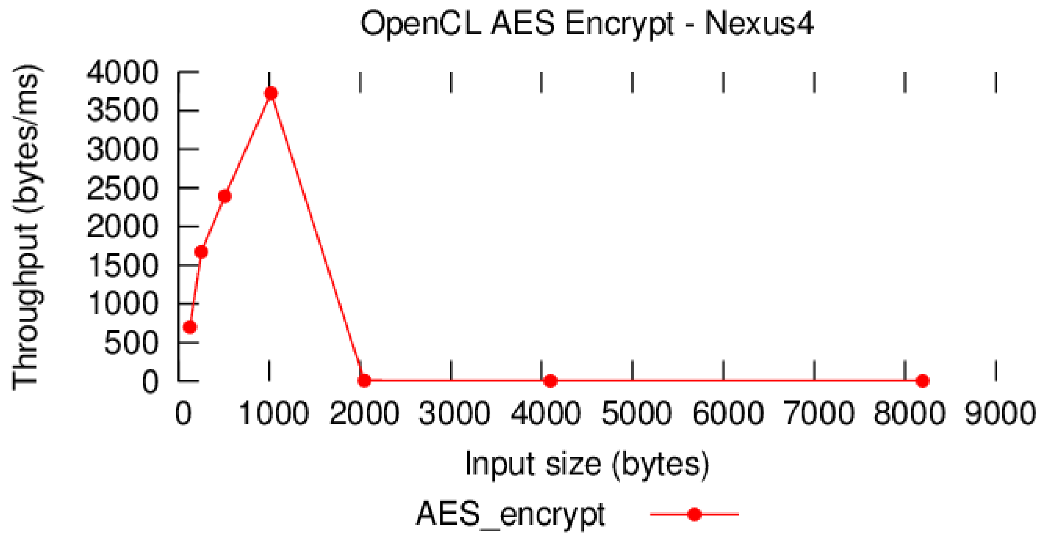

Figure 1: AES encryption throughput for the Nexus 4

From figure 1, we can see that at encryption sizes of 128, 256, 512, and 1024 bytes, we get increasingly better throughput, with a maximum throughput of 3723.64 bytes/ms ~ 3.64 KB/ms. However, there is a large drop-off (down to 8.25 bytes/ms) in throughput going from 1024 bytes to 2048 bytes, with the lowest throughput at 4.18 bytes/ms for 8192 bytes. This is a very unexpected result, and warranted further investigation into how the OpenCL kernel was being executed, which we will discuss in subsequent sections

Limitations and further investigation of instability:
As shown in figure 1, we were only able to collect measurements of AES encryption throughput for input sizes of up to 8K, since sufficiently long running instances of an OpenCL kernel would cause the device to be forefully rebooted after approximately 10 seconds from the program beginning execution. This behaviour was further investigated using OpenCL programs that execute infinite loops (so rule out that the behaviour was characteristic of a bug in the OpenCL kernel). However, the behaviour persisted even in trivial programs.

We attempted to disable OpenGL on the device to rule out that some other process is using the GPU for background activity (i.e. when the display is inactive). We attempted to disable OpenGL via Linux kernel compiler flags. In particular, we disabled the MSM_KGSL flag whose description is "3D graphics driver. Required to use hardware accelerated OpenGL ES 2.0 and 1.1." While disabling this flag seemed to disable OpenGL (since Android processes that use it did not start), it also disabled OpenCL, since an internal (in a proprietary library) assertion error message gets printed upon calling OpenCL commands.

Hence, instead of this approach, we took the simpler approach of removing shared libraries from /system/lib that are OpenGL specific [cite embedded android]. This caused the Android processes that typically use OpenGL not to start. However, even in this scenario, the device still forcefully reboots on sufficiently long OpenCL programs with a similar timeout period of 10 seconds.

Next, we investigated whether this behaviour may be some kind of explicit hang detection, based on the available GPU driver code for the Adreno 320 in the Nexus 4 kernel source tree. In particular, we looked at the code residing in the drivers/gpu subdirectory of the source tree for any mention of a timeout. adreno.c defines a adreno_waittimestamp function that polls hang_detect_regs, an array of register addresses corresponding to the names:

- A3XX_RBBM_STATUS
- REG_CP_RB_RPTR
- REG_CP_IB1_BASE
- REG_CP_IB1_BUFSZ
- REG_CP_IB2_BASE
- REG_CP_IB2_BUFSZ

If all the registers do not change after KGSL_TIMEOUT_PART (defined as 2 seconds), then a GPU device hang is determined (by adreno_hang_detect in adreno.c). Unfortunately, none of these registers are documented as to what values they contain. Furthermore, the timeout period is much smaller than the observed 10 second timeout period. Also, if the hang detection was expected behaviour, the expected behaviour would be to terminate the process occupying the GPU and restart the entire device. Hence, we decided not to pursue using OpenCL on the Nexus 4 any further due to instabilities, and instead investigate another phone model, the MotoX.

Investigating the OpenCL programming model as it relates to the AES algorithm implementation:
After inspecting the code for running the OpenCL encryption algorithm, it's clear that the code was not being called with the proper arguments, which may explain the degradation in throughput for larger input sizes as observed on the Nexus 4 (figure 1).

In particular, we investigated more about the OpenCL programming model. The OpenCL programming model defines work-items to be an executing instance of an OpenCL kernel (assigned a local id), and work-groups to be equally sized groups of work-items (assigned a global id). The OpenCL model ensures that work-items within a work-group will execute concurrently, but provides no guarantees whether work-groups will execute concurrently or serially. When you execute an OpenCL kernel, you specify the number of work groups, and the number of work items with each work group. All work-items within a group each get a unique local id, but they all get the same global id (unique from other groups). The work-items then make use of their local and global ids to appropriately divide up the work, and can make use of synchronization constructs like a work-group barrier to synchronize their execution (but only for work-items within the same group) [4].

The OpenCL encrypt kernel [2] is implemented using a data parallel model. In particular, it only makes use of the global id of a work-group to index itself into the input array at a particular position (e.g. Index 0 of the input array), with one work-group instance being responsible for encrypting an entry of the input array. The kernel also makes use of the vector instruction set of the GPU to increase the number of bytes that a single "entry" in the input array corresponds to [3]. In particular, in addition to traditional analogs to C data types like uint (a 32-bit integer for a total of 4 bytes), there are vector data types like uint4 (4 32-bit integers, for total of 16 bytes). Using these vector types, a single work-item can operate on 16 bytes at once, whereas a traditional CPU would operate on 4 bytes at once. Thus, given an input array of N bytes, we only need N/16 work-groups of 1 work-item each to encrypt an input array. Hence, in this implementation, the local id is ignored, since a work-group only consists of a single work-item.

The error we found in the code was that multiple work-items were being allocated to a group, with the number of work-groups being equal to the input array size. This means that work-items were encrypting entries multiple times (arbitrairily chosen as the maximum number of concurrent work-items available). We corrected this to use 1 work-item, and N/16 work-groups. Another issue that was resolved was that the input array was being allocated on the stack rather than the heap, which would cause a segmentation fault to occur for a input size of only 8MB (though this limitation was not observed in the Nexus 4 since the device would reboot before reaching this point).

AES encryption on the MotoX:

We switched to using the MotoX, which uses Android 4.2 and has the OpenCL libraries. We immediately discovered that the issue of the device forcefully rebooting for sufficiently long running OpenCL kernels was gone. Thus, we conducted the OpenCL encryption measurements on larger input sizes. Since the implementation uses input arrays that are a multiple of 16 bytes, we started with 16 bytes and incremented by powers of two up to 512 MB, for total of 26 samples.

Figure 2: AES encryption throughput for the MotoX

The summary statistics for the throughput measurements in figure 2 are provided in table 3.

Minimum	**First Quartile**	**Median**	**Mean**	**Third Quartile**	**Maximum**
58.18	2681.36	7480.30	5935.56	9036.79	9266.89

Table 3: Summary statistics for AES encryption throughput (in bytes/ms) on the MotoX

From figure 2, we see that for initial input sizes of 16 bytes up to 4MB (which encompasses the range of throughput on the y-axis from 0 bytes/ms to just less than 9000 bytes/ms), we get increasing gains in throughput. For input sizes larger than 4MB (points just above 9000 bytes/ms), our gains in throughput level off around 9000 bytes/ms, and reach a maximum of throughput of 9266.89 bytes/ms. That is, we are able to achieve a maximum encryption throughput of 8.84MB/second at an input size of 512MB. Given an memory of 2GB, it would take 231.67 seconds (3 minutes and 52 seconds) to encrypt the entire device memory at this throughput.

Achievements for the class presentation (November 27) and final report (December 13):

Further investigate the OpenCL programming model and parallelism in the AES implementation

As discussed above, we determined that the OpenCL AES implementation spawns N/16 (where N is the size in bytes of the input to encrypt) separate instances of the OpenCL kernel, and that these use the work-group identifier to parallelize operation on the input, thus using a work-item of size 1. However, it is not clear if this is an optimal way to divide up the work, since it is possible that simply using fewer work-groups which operate over more entries of the input could yield faster throughput. Thus, we need to run the OpenCL kernel under different operating parameters, and possibly run it on a desktop machine so as to compare our performance with previous results in literature.

Benchmark battery consumption (amplitude during execution) and throughput (rate of encryption) on the MotoX

Once we have determined an optimal implementation of the OpenCL encryption algorithm, we intend to provide final benchmark results and measurements of battery consumption relative to idle energy levels of the device.

References:

[1] Che, Shuai, et al. "Rodinia: A benchmark suite for heterogeneous computing." Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on. IEEE, 2009.

[2] Github, *OpenCL-AES*, https://github.com/softboysxp/OpenCL-AES

[3] Munshi, Aaftab, et al. *OpenCL programming guide*. Pearson Education, 2012.

[4] Yaghmour, Karim. *Embedded Android: Porting, Extending, and Customizing*. O'Reilly, 2013.