Alethea Butler

Chloe Eghtebas

SoftSys 2014

GPU Exploration: Breaking Hashes

We explored GPU programming though the parallelization of a brute-force attack against SHA-256 hashes. We successfully built the brute force hash breaking on host then ported it to the device. In the end, we had a hash-breaking algorithm that was able to run on device not much more quickly than on host due to our parallelization methods.

We started off implementing SHA-256 hash using the NIST spec for all of the SHA family (http://csrc.nist.gov/groups/STM/cavp/documents/shs/sha256-384-512.pdf) in C, so we could easily debug it and later port it to OpenCL.

Next, we got some sample OpenCL code working running on our computers and confirmed that we are computing using the GPU (sample code could be downloaded here: http://twimgs.com/ddj/images/article/2011/0711/add_numbers.zip). After, we delved into understanding OpenCL objects and their structure with (https://www.khronos.org/opencl/) inorder to create a program that reads in a kernel. Below is a description of how a typical OpenCL main function for someone new to OpenCL.

At this point, we started seriously thinking about how to optimize the brute force algorithm. Sweeping through the space of strings alphabetically would find strings like `“aaa”` faster than it would strings similar to common passwords. We wrote a python script to parse a list of common passwords and generate a C file with a lookup table for what characters commonly follow each other. This allows us to bias our sweep so that it will encounter `“password”` before `“aposfgj”`, reducing the mean time to a match on real-world password data.

What goes inside OpenCL main.c

First step in making our own OpenCL code was to get platform and device ID’s using the `clGetPlatformIDs` and `clGetDeviceIDs`. They return a value or a list of the platform or devices found respectively. Based on the platform found, `clGetDeviceIDs` will accept flag options like `CL_DEVICE_TYPE_GPU` in order to limit the devices returned to GPUs in this example.

Next, an OpenCL context object is required. A context is an OpenCL object in which is responsible for executing the kernels and where synchronization and memory management are defined. Synchronization is done with CommandQueues and memory management is done with MemObjects.

The CommandQueue will get filled with tasks to send to the GPU. To create a CommandQueue we pass in both, the context and device id into `clCreateCommandQueue`. The default command queue is a first-in first-out priority queue and one of the options of `clCreateCommandQueue` is a flag which allows to change the priority of the tasks.

The MemObject allows for shared memory between the host and the device. To create a memory object, we pass in context, a flag specifying what type of memory it is (i.e. `CL_MEM_READ_WRITE`), the size of the MemObject we want to create, and the value the memory will be initialized to.

Next, the program is created either with source or with a binary. Then kernel is built. The kernel is a separate piece of .cl code. Once, the kernel is created, we pass in the various arguments to the specific kernel. The kernel with set arguments are then enqueued into the command queue.Lastly, the results from the task are then read from the buffer or memory.

This described the flow between OpenCL object which can also be seen in the below flow diagram. We didn't need some of the other objects shown below for our application. Additionally, there is a way to control the number of work units and work units called using `clEnqueueNDRangeKernel`.

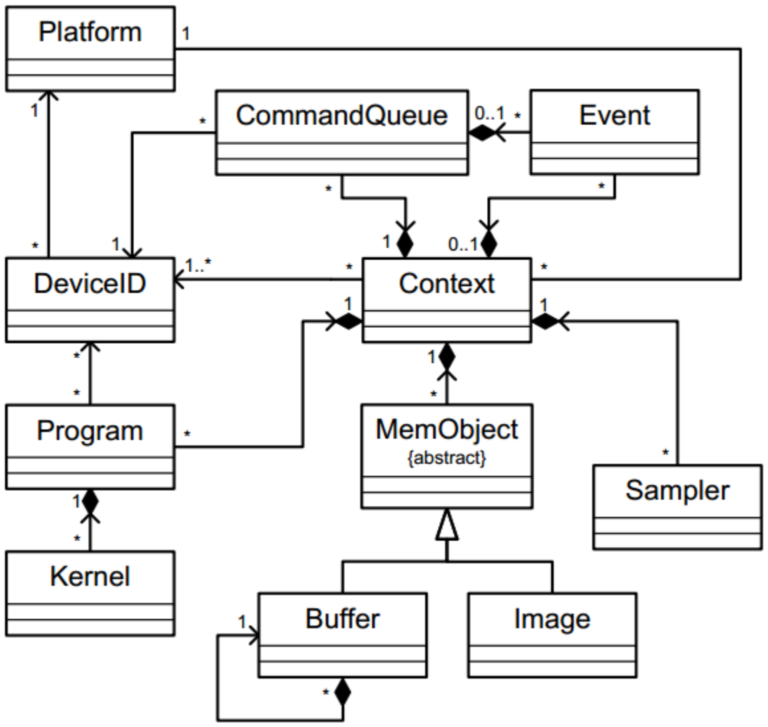

Figure from OpenCl Specification document (http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf)

Back to Hash Breaking

Then we ported our implementation of SHA-256 into an OpenCL kernel. Given more time, we would have been able to refactor this kernel into part of a distributed hash breaking system, but we ran out of time to fully implement the OpenCL synchronization patterns necessary to insure that each kernel got exactly one string to hash and the entire search space was covered. We did succeed in creating a proof of concept using synchronous CPU code without GPU acceleration.

We feel that we have achieved our learning goals. In particular, we both understand what the GPU is capable of and what future applications could benefit from GPU optimization. In addition, we now understand the basics of how computing on a GPU works using the powerful OpenCL library. Also, we both got exposure to the field of cryptosecurity which was a large overlap in our interests. We still feel that more experience with designing the synchronization of the host code's interaction with the device and inter kernel synchronization would be valuable before going out and coding for GPU's.

Our deliverable can be found in either of our softsys repo's under hw06. Here, you will find two directories with a readme on how to install the dependencies and how to run our code. The first directory is called CPU. Under this directory you will find our application working without GPU. The second directory is called GPU. Here is code which uses the GPU to hash a string.