FIR filtering and AES encryption with OpenCL 2.0

Carter McCardwell, Tuan Dao, Saoni Mukherjee, David Kaeli
Dept. of Electrical and Computer Engineering
Northeastern University
Boston, MA

***Abstract*—OpenCL has become a popular standard to leverage the unique power/performance opportunities found on heterogeneous systems. In this short contribution, we evaluate the latest parallel programming features supported in the OpenCL 2.0 standard. We explore using shared virtual memory and dynamic parallelism to accelerate two example applications.**

I. Introduction

As we see new heterogeneous architectures appear on the market, supporting programming frameworks that can exploit new classes of architecture are also starting to appear. In this paper we explore OpenCL 2.0 by leveraging a number of its new features to accelerate two popular applications: 1) Finite Impulse Response filtering, and 2) the AES-128 encryption standard. We will begin by discussing the latest features in OpenCL 2.0, and then describe how our two sample applications can leverage these features.

II. OpenCL 2.0

OpenCL is a programming/runtime framework to enable applications to execute across heterogeneous platforms [1]. OpenCL is presently supported to run on a number CPUs, graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other devices. In 2013, Khronos released OpenCL 2.0 [2], and announced a number of new and novel programming features. Some of these features include:

- Dynamic Parallelism: This allows device kernels to enqueue child kernels to the same device with no host interaction. This feature will enhance the performance of programs that feature multiple kernels, support recursive execution, as well as provide flexibility during the design of new applications.
- Shared Virtual Memory: This feature allows the host and the device to share a common virtual memory system, enabling pointers to be shared between host and device. This can help eliminate unnecessary RAM usage and simplify application code. This feature will be leveraged extensively throughout our Finite Impulse Response (FIR) application.
- Images support: OpenCL 2.0 allows sRGB and 3D image support. This can help with digital imaging applications, easing the handling of images and image streams.
- Android Installable Client Driver Extension: Android is the most popular mobile operating system today, so this feature provides an advantage for OpenCL. It allows OpenCL implementations to be discovered and loaded as a shared object on Android systems.
- Generic Address Space: Functions can be written without specifying a named address space for arguments, especially useful for those arguments that are declared to be a pointer, pointing to a type, eliminating the need for multiple functions to be written for each named address space used in an application.

In the next section, we will discuss how to use some of these new features to improve code readability and performance of OpenCL application.

III. FIR filtering with OpenCL 2.0

A Filter Impulse Response (FIR) filter produces an impulse response of finite duration [3]. The impulse response is the response to any finite-length input. The FIR filtering program is designed to have the host send array data to the FIR kernel on the OpenCL device. Then the FIR is calculated on the device, and the result transferred back to the host.

When designing the FIR application in earlier versions of OpenCL (e.g., version 1.2), an input array of cl_float pointers were allocated using a malloc call. The output, coefficients and a temporary output array were similarly declared. The host program then reads the data from an input file into the input array. The host will also allocate cl_mem memory buffers for the input, output and temporary output data using clCreateBuffer. After copying all the data from the input array into the input buffer, the program runs the kernel and the computation begins. The results are then written into an output buffer, which is copied into the output array, and then control is returned to the host.

Moving to OpenCL 2.0, the code can be greatly improved. The input array is now allocated using $clSVMAlloc$:

$$\begin{aligned}&input = (cl\_float*)clSVMAlloc\\&\quad(context, CL\_MEM\_READ\_WRITE,\\&\quad numTotalData^{*}sizeof(cl\_float), 0);\end{aligned}$$

Since we use the same variable type as before, there is no need to rewrite the code when moving to OpenCL 2.0. The other arrays are similarly declared. However, since $clSVMAlloc$ allows the allocated memory to be shared between the host and the device, there is no need to allocate input buffers anymore, thus eliminating time-consuming $clCreateBuffer$ and $clEnqueueWriteBuffer$ calls, reducing memory usage, and reducing the execution time. The kernel arguments are set using $clSetKernelArgSVMPointer$ since we are now using SVM pointers. The rest of the program requires minimal modifications. In a nutshell, by using SVM pointers, the programmer no longer has to copy the input and output data to and from buffers, thus reducing memory usage, reducing the execution time, and reducing the burden on the OpenCL programmer.

IV. AES-128 WITH OPENCL 2.0

Our second application is an implementation of the Advanced Encryption Standard (AES) [4]. The program reads an input file and encrypts it with a given encryption key. In this application, we can leverage the new features of OpenCL 2.0 using shared virtual memory and dynamic parallelism (i.e., kernel enqueuing).

The shared memory scheme can greatly benefit a program such as encryption, since the runtime does not need to spend a significant amount of time copying blocks of data back and forth to the device. Data is placed into the shared memory where it is copied by each work-unit and/or kernel after processing has finished. This benefit becomes more important in the context of dynamic parallelism, where a controller kernel automatically launches device kernels using the data in shared memory. Hence, the device can spend more time processing, and potentially improve task-level parallelism.

The host operation simply focuses on reading data into/out of shared memory, while the controller kernel can delegate work to slave kernels. Communication between the host and the controller kernel can be accomplished using flags in shared memory. Instead of using the default OpenCL queue to send events, pointer arrays that hold dynamically-updated memory addresses, are sent to the controller kernel. The pointer array will address the flags that signal when new data has been copied by the host, and will include the memory address of the new data. For example, the AES algorithm requires that data be broken into 16-byte *states* that are processed individually.

Before the host commits the data into shared memory, each state is aligned and unionized with a sequence number and a Boolean value that is set to 0, before the state is processed. Since the data is aligned on a fixed number of bytes, the kernel and host can read the memory space using pointer arithmetic. After the kernel processes the state, it will set the corresponding flags to 1. The host uses the sequence number to copy the data back in the correct order. The data does not need to be processed in a specific order, so the controller kernel can adapt and scale the number of workunits based on the compute capabilities of the device, and also based on the size of the plaintext file. In most cases, the bottleneck in this application becomes the speed of reading the initial file and writing the final file.

Using OpenCL 2.0's shared virtual memory and dynamic parallelism, we have the ability to create more flexible and dynamic applications. In the case of the AES example, the new simultaneous host/kernel operation is much more efficient. So here we have three benchmark programs that are being run for the test of the increased speed of using shared virtual memory (SVM). First, there is a multithreaded CPU implementation (AES_cpu) that will be used as a baseline point. For speed-up, we have implemented the same in OpenCL 1.2 ($AES_ocl1.2$) and then to take advantage of OpenCL 2.0's SVM feature, we have implemented it in OpenCL 2.0 ($AES_ocl2.0$). In this case, as mentioned earlier, instead of copying the data to global memory, we use SVM. Both versions of the GPU program use the CPU for key expansion, since the CPU can expand the private key and avoid the overhead of kernel start. The expanded key can then be dynamically placed in memory as input to the OpenCL kernel so that the keys will exist in constant memory. When each kernel starts, each state (i.e., AES data block) is copied into the register of the work-unit, processed, and then written back to global memory.

V. RESULTS

Data for AES: The data tested is a 600kb copy of Sherlock Holmes, and 10, 50, 100, and 1000-megabyte files consisting of random data.

Datasize	AES_{cpu}	$AES_{ocl1.2}$
0.6 MB	1271 ms	806 ms
10 MB	19946 ms	12270 ms
52 MB	109275 ms	59080 ms
105 MB	2117082 ms	116783 ms
1 GB	1160867 ms	2117082 ms

VI. SUMMARY

In this contribution we have discussed the benefits of OpenCL 2.0. We have explored some of the benefits of this new programming framework by discussing two applications: FIR filtering and AES-128 encryption. In our future work, we will consider programmer productivity measures that we positively impacted when leveraging OpenCL 2.0.

REFERENCES

[1] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[2] K. O. W. Group *et al.*, "Opencl 2.0 specification," *Khronos Group, Nov*, 2013.

[3] A. V. Oppenheim, A. S. Willsky, and S. H. Nawab, *Signals and systems*. Prentice-Hall Englewood Cliffs, NJ, 1983, vol. 2.

[4] J. Daemen and V. Rijmen, *The design of Rijndael: AES-the advanced encryption standard*. Springer, 2002.