Exploring the Features of OpenCL 2.0

Saoni Mukherjee, Xiang Gong, Leiming Yu, Carter McCardwell, Yash Ukidave, Tuan Dao,
Fanny Nina Paravecino, and David Kaeli
{saoni, xgong, ylm, cmccardw, yukidave, tdao, fninaparavecino, kaeli}@ece.neu.edu
Department of Electrical and Computer Engineering, Northeastern University, Boston, MA

I. Introduction

The growth in demand for heterogeneous accelerators has stimulated the development of cutting-edge features in newer accelerators. The heterogeneous programming frameworks such as OpenCL have matured over the years and introduced new software features for developers. The first version of OpenCL (1.0), was a basic programming model. The next versions, 1.1 and 1.2, enabled different memory management techniques, and gave users more fine-grain control over the resources. With version 2.0, OpenCL has undergone a significant evolution by introducing features that can exploit emerging hardware capabilities as well as provide users more ease of programming and control over handling the resources.

In this paper we explore one of these programming frameworks, OpenCL 2.0. To drive our study, we consider a number of new features in OpenCL 2.0 using four popular applications from a range of computing domains including signal processing, cybersecurity and machine learning. These applications include: 1) the AES-128 encryption standard, 2) Finite Impulse Response filtering, 3) Infinite Impulse Response filtering, and 4) Hidden Markov model. We will begin by discussing the latest runtime features enabled in OpenCL 2.0, and then discuss how well our sample applications can benefit from some of these features.

II. OpenCL 2.0

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

- **Dynamic Parallelism:** It allows kernels to launch other kernels (child) without any host interaction saving both time and resources, and the CPU can be busy doing other computations. This will improve the performance of applications with multiple kernels, help implement recursive executions, and will also provide the user more flexibility while designing the applications.
- **Shared Virtual Memory:** It is one of the landmark features in OpenCL 2.0. In this version, the host and the device kernel can share a common virtual address range, which enables pointers to be shared between host and device. So explicit copies from host-to-device and device-to-host can be eliminated. This feature will be leveraged extensively throughout our implementation of different applications.
- **Image support:** OpenCL 2.0 enables support for sRGB and 3D image. This is particularly beneficial to the digital imaging applications. We are not using this feature at this time.
- **Android Installable Client Driver Extension:** This version adds a new feature, where OpenCL implementations that are non-native to Android can be used as a shared object on Android systems.
- **Generic Address Space:** Earlier during declaring a pointer or using it as an argument to a function, the pointer had to be specified with an address space that the pointer is pointing to. In this version, it stays in the private address space and it can point to anything on the named address spaces inside the generic address space.

In the next sections, we will the applications we have chosen to explore some of the features in OpenCL 2.0, and discuss how we optimize these applications using OpenCL 2.0.

III. Applications

We introduce four applications from a variety of domains ranging from signal processing to cybersecurity applications. We start with this small set to demonstrate how we can leverage some of the features described above. We are actively working on a more complete set, that will support both OpenCL 2.0 and Heterogeneous Systems Architecture [3] features. We will discuss the entire set as part of our presentation.

A. Advanced Encryption Standard

Our first application is an implementation of the Advanced Encryption Standard (AES). The program takes plaintext as input and encrypts it using a given encryption key [4]. AES

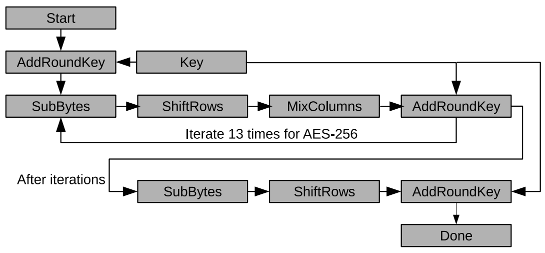


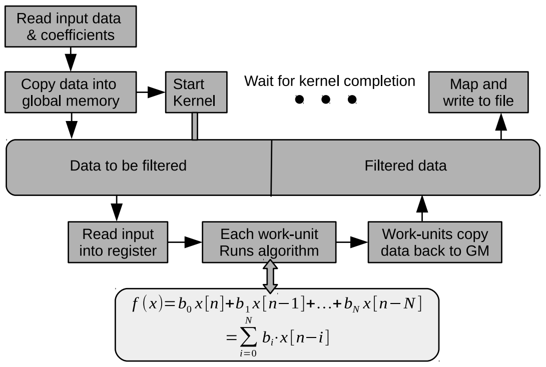
Fig. 1: AES algorithm

has been adopted by the US government to encrypt data in applications ranging from personal to highly confidential domains [5].

AES has a fixed block size of 128 bits. The key size can vary between 128, 192 and 256 bits. Our implementation uses a key size of 256 bits. A block is the unit of plaintext that the algorithm takes as input and uses to produce the corresponding n-bit cipher text. Text that is longer than a block is divided into multiple blocks, the last chunk is padded, and each block is encrypted separately. The AES algorithm is comprised of many rounds, as shown in Figure 1, that ultimately turn plaintext into cipher-text. Each round has multiple processing steps that include AddRoundKey, SubBytes, ShiftRows and MixColumns. Key bits must be expanded using a precise key expansion schedule.

B. Finite Impulse Response

A Finite Impulse Response (FIR) filter produces an impulse response of finite duration [6]. 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 filter is calculated on the device, and the result is transferred back to the host.



Read input data & coefficients
Copy data into global memory
Start Kernel
Wait for kernel completion
Map and write to file
Data to be filtered
Filtered data
Read input into register
Each work-unit Runs algorithm
Work-units copy data back to GM
$f(x)=b_0x[n]+b_1x[n-1]+\ldots+b_Nx[n-N] = \sum_{i=0}^{N} b_i \cdot x[n-i]$

Fig. 2: GPU implementation of a FIR filter.

Given an input signal $x[n]$, the FIR produces $f(x)$ as the output signal. For an Nth order filter there are $(N+1)$ terms on the right hand side of the equation shown in Figure 2. Then a weighted sum of the input signal and b_i is computed. b_i is the value of the impulse response at the ith instant, where $0 \leq i \leq N$ in an N-th order filter.

C. Infinite Response Filter

An Infinite Impulse Response (IIR) filter requires less processing power than a Finite Impulse Response (FIR) filter for the same design requirements. IIR can be further decomposed into multiple parallel second-order IIR filters to achieve better performance [7]. We present our multichannel version of a parallel IIR filter and implement it on the GPU using the OpenCL framework. The transfer function used to compute the parallel IIR output is shown in Equation 1. Our parallel IIR implementation for multiple channels is illustrated in Figure 3.

$$H^z(z) = c_0 + \sum_{i=1}^{N_1} \frac{f_i}{1+e_i z^{-1}} + \sum_{i=1}^{N_2} \frac{f_{N_1+2i-1} + f_{N_1+2i}z^{-1}}{1+e_{N_1+2i-1}z^{-1} + e_{N_1+2i}z^{-2}} \tag{1}$$

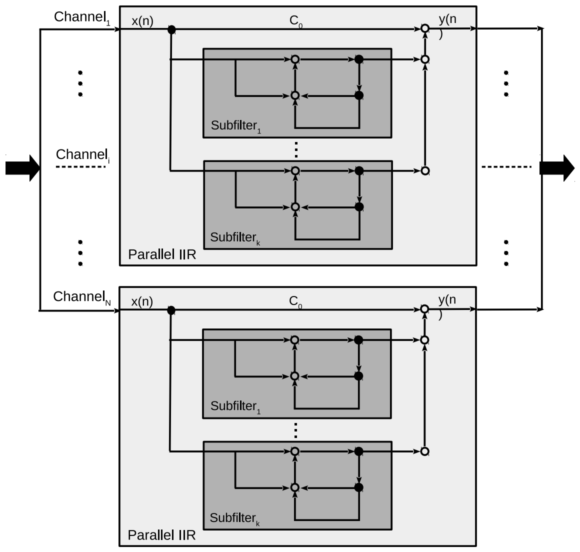


Fig. 3: Multichannel parallel IIR

D. Hidden Markov Model

A Hidden Markov Model (HMM) is a static Markov model that can generate probabilistic meaning without knowing the hidden states [8]. It explores the relationship between the hidden states and observations during the Markov process. We have developed a Hidden Markov Model on GPU that targets isolated word recognition [9]. In order to achieve the best performance on the GPU device, we express the data-level and thread-level parallelism in the HMM algorithm.

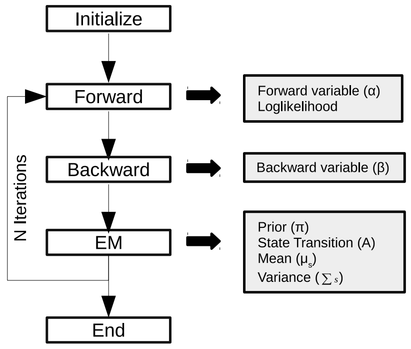


Fig. 4: The HMM algorithm used in this work.

In HMM, as shown in Figure 4, the computation is expressed in matrix form operation to obtain the coalescing and computation efficiency. The $loglikelihood$ and Forward variable, α, are computed at the Forward stage. At the Backward stage, the Backward variable, β, is populated. To update the expected values, such as prior, hidden states, mean and covariance, Expectation and Maximization (EM) is applied [10]. For this application, matrix-vector, matrix multiplication and parallel reduction dominate execution in our GPU implementation [11].

IV. Implementation

First, these four applications are implemented in OpenCL 1.2 – we will treat this implementation as a baseline. Next, the same four applications are implemented in OpenCL 2.0 leveraging its new features. In this section we discuss some of the implementation details for the four applications.

The easiest OpenCL 2.0 feature to leverage is Shared Virtual Memory. Any application that requires multiple memory copies can benefit from the new coarse-grained SVM support provided in OpenCL 2.0. We avoid explicit copies from host-to-device and device-to-host, when using SVM. The SVM mechanism maps data into a contiguous block of memory accessed through a host accessible pointer and provides the user a software abstraction from all the memory management. It waits until the unmap command to allow a kernel-instance to safely read and/or write the buffer. Larger the file, more pronounced is the effect of using SVM.

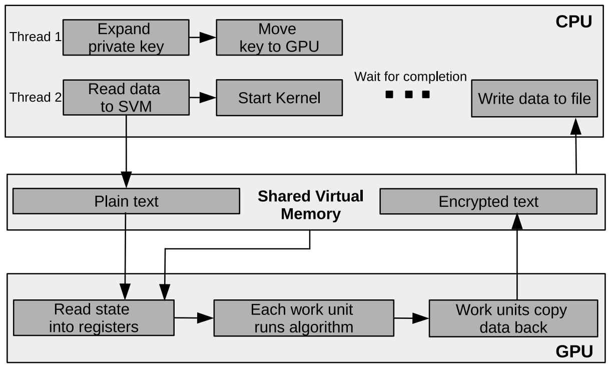


Fig. 5: The AES GPU implementation used in this work.

In AES, both versions of the GPU program use the CPU for key expansion. Our experiments reveal that it is faster to use the CPU for key expansion. The expanded key is then dynamically placed in memory as input to the OpenCL kernel so that the keys will exist in constant memory, as shown in Figure 5. When the kernel starts, each state (i.e., AES data block) is copied into the register of the work-unit, processed, and then written back to the global memory. In signal processing applications FIR and IIR, we have used the SVM to keep track of the input signal and its coefficients. When these are later needed to calculate the output signal, which is a weighted sum of the coefficients and input signal, instead of copying the data coefficients, input, and partial sum back and forth between host and device, they are managed through SVM. SVM maps it into a contiguous block of memory accessed through a host accessible pointer. It waits till the unmap command to allow a kernel-instance to safely read and/or write the buffer.

Another new OpenCL 2.0 feature that we have used is dynamic parallelism. In HMM, since updating the expected values for each hidden state is an indepedent operation, there is a golden opportunity to explore dynamic parallelism. The dynamic parallelism feature allows the kernel to launch the $relabelUnrollKernel$ function from within the main kernel. This saves time as control does not need to be handed back to the host to start the next kernel. This saves many memory copy operations and kernel launch startup overhead. Together with this, SVM helps in avoiding to work with explicit memory copy operations. The data will be stored in one pool of memory until the algorithm is finished.

V. Results

A. Evaluation Platform

To drive our study, we use an AMD Radeon R9 290x GPU. We evaluate the four applications developed in both OpenCL 1.2 and 2.0. Table I includes the particulars of the GPU platform used for evaluating OpenCL 2.0 features, as compared to our baseline. For OpenCL 2.0, the AMD OpenCL 2.0 beta driver version 14.41 has been used, since during the time of development of this paper, the final version was not released.

TABLE I: Architectural specifications of the GPU used for evaluation.

Peak Single Precision FLOPS(Gflops)	5632
Peak Bandwidth (GB/s)	352
Streaming Cores	2816
Clock Rate (MHz)	1000
Global Memory (GB)	8
Wavefront Size	64

B. Performance Analysis

All of the applications presented enjoy some benefits when using SVM. The degree of speedup depends on amount of data size being managed using SVM and resulting a number of copy operations avoided. In figs. 6, 7 and 9, the runtimes for IIR, FIR, HMM and AES are compared with baseline, for a range of input sizes. All of the runtimes reported include all communication overhead experienced by the application, including preprocessing, data transfer and kernel runtime.

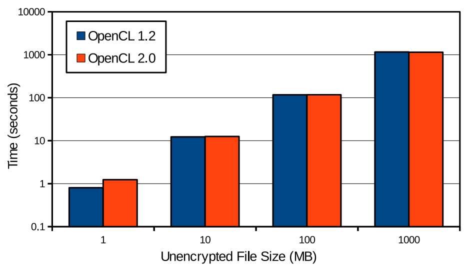


Fig. 6: Comparison of runtimes of AES in OpenCL 1.2 and 2.0.

Previous work has shown that moving the data from and to the device memory may be the bottleneck in performing AES on the GPU [12]. So, we first explored speedup using SVM, and then further explored optimizations using dynamic parallelism. The data is divided into 16-byte blocks, called *states* which are encrypted completely in parallel. We obtain speedup with SVM, but do not observe similar speedups when using dynamic parallelism. The child kernels are launched for memory intensive SubBytes and ShiftRows processing, when the parent kernel is active on the GPU. The child kernels

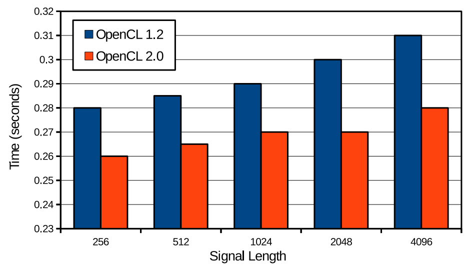


Fig. 7: Comparison of runtimes of FIR in OpenCL 1.2 and 2.0.

are restricted by the amount of available compute resource, thereby leading to decreased occupancy for child kernels. This increases the latency of child kernels and leads to decrease in the overall throughput of the application.

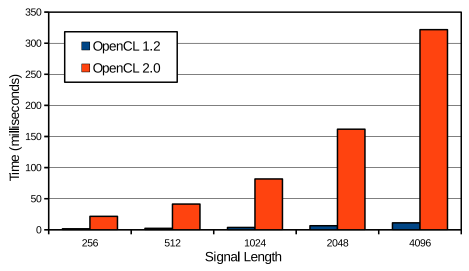


Fig. 8: Comparison of runtimes of IIR in OpenCL 1.2 and 2.0.

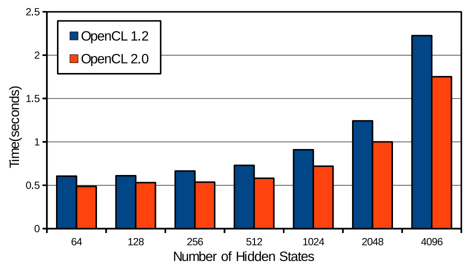


Fig. 9: Comparison of runtimes of HMM in OpenCL 1.2 and 2.0.

Another interesting observation is that when two identical kernels of the two signal processing applications- IIR and FIR with same input are run with OpenCL 1.2 and OpenCL 2.0, the kernel that is run with OpenCL 2.0 is faster.

For HMM, we have experimented with another OpenCL 2.0 feature- work-group function. However, utilizing this feature resulted in a slowdown for the IIR application compared to the same algorithm implemented in OpenCL 1.2, as shown in Figure 8. The HMM application computes a weighted sum of input signals. Work-group function is used to perform reduction over the weighted input signals to compute the sum. To the best of our knowledge, there was no code profiler available that was compatible with the current version of the driver at the time of this work. Presently we are moving to the next version of the OpenCL 2.0 driver that will be supported by CodeXL. We will provide a full explanation of this issue in the final version of this paper.

VI. Conclusion

In this contribution we have explored the benefits of OpenCL 2.0. We have discussed some of the benefits of this new programming framework using four applications: AES-256 encryption, FIR and IIR filtering, and a HMM. During implementation, we have found OpenCL 2.0 features (SVM) that provide consistent benefits. We also found issues with utilizing the work-group function. OpenCL 2.0 has the potential to generate faster running programs and increase programmer productivity.

VII. Future Work

In the near future, we would like to add fine-grained SVM to the existing implementations. Fine-grained SVM allows pointers to be directly shared between the host and device, hence memory can be written to and read from simultaneously. This will enable other features such as dynamic parallelism to be added too. Then the host operation can simply focus on reading data into/out of shared memory, while a 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. We are also interested in exploring other features introduced in OpenCL 2.0, such as Image support, in the future.

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] Khronos OpenCL Working Group and others, "OpenCL 2.0 Specification," *Khronos Group, Nov*, 2013.

[3] G. Kyriazis, "Heterogeneous system architecture: A technical review," *AMD Fusion Developer Summit*, 2012.

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

[5] N. F. Pub, "197: Advanced encryption standard (AES)," *Federal Information Processing Standards Publication*, vol. 197, pp. 441–0311, 2001.

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

[7] B. Porat, *A course in digital signal processing*. Wiley New York, 1997, vol. 1.

[8] L. E. Baum and T. Petrie, "Statistical inference for probabilistic functions of finite state Markov chains," *The annals of mathematical statistics*, pp. 1554–1563, 1966.

[9] L. R. Rabiner, S. E. Levinson, and M. M. Sondhi, "On the application of vector quantization and hidden Markov models to speaker-independent, isolated word recognition," *Bell System Technical Journal, The*, vol. 62, no. 4, pp. 1075–1105, 1983.

[10] T. K. Moon, "The expectation-maximization algorithm," *Signal processing magazine, IEEE*, vol. 13, no. 6, pp. 47–60, 1996.

[11] L. Yu, Y. Ukidave, and D. Kaeli, "GPU-accelerated HMM for Speech recognition," in *Heterogeneous and Unconventional Cluster Architectures and Applications Workshop (HUCAA14). IEEE*, 2014.

[12] S. A. Manavski, "Cuda compatible gpu as an efficient hardware accelerator for aes cryptography," in *Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on*. IEEE, 2007, pp. 65–68.