USING OPENCL TO INCREASE SCA APPLICATION PORTABILITY

Steve Bernier (NordiaSoft, Gatineau, Québec, Canada; Steve.Bernier@NordiaSoft.com);
François Lévesque (NordiaSoft, Gatineau, Québec, Canada;
Francois.Levesque@NordiaSoft.com);
Martin Phisel (NordiaSoft, Gatineau, Québec, Canada; Martin.Phisel@NordiaSoft.com);
David Hagood (Aeroflex, Wichita, Kansas, USA; David.Hagood@Aeroflex.com);

ABSTRACT

The Software Communications Architecture (SCA) is the defacto standard to build Software Defined Radio (SDR) radios. Over one hundred thousand SCA military radios have been deployed worldwide by several nations. The SCA offers a component-based operating environment for heterogeneous embedded system that ensures applications are portable across platforms made of General Purpose Processors (GPPs) and Digital Signal Processors (DSPs).

The SCA offers a high level of portability for applications have been implemented for GPPs and DSPs. SCA components can easily be ported across different processors using different operating systems and communication buses. However, the level of portability is reduced when source code is tuned for specific instructions sets. Furthermore, using Field Programmable Gate Arrays (FPGAs) drastically reduces the level of portability for SCA components.

Specialized instruction sets are very widely used for high performance military radio platforms. Consequently, finding a solution to increase portability of components that run on such processing elements could provide significant cost reductions when an application is ported. In fact, application portability is the number one innovation on the top ten list of most wanted innovations compiled by the Wireless Innovation Forum (WInnF).

This paper describes how the Open Computing Language (OpenCL) can be used in conjunction with the SCA to build more portable applications. OpenCL is a framework for writing programs that execute across heterogeneous platforms consisting of GPPs, DSPs, FPGAs, and graphics processing units (GPUs). The paper starts with an overview of OpenCL, describes how SCA components can be built using OpenCL, provides performance metrics, and concludes on how the SCA could be improved to offer better support for OpenCL.

1. INTRODUCTION

The SCA was created to standardize how real-time embedded applications are implemented, packaged, installed, deployed, and controlled. The main goal of the SCA is to make applications very portable across different heterogeneous systems. It was created for the Joint Tactical Radio System (JTRS) program, a US DoD program that funded the development of a new kind of military radios: Software-Defined Radios (SDRs). The JTRS program started by funding the definition of a new standard called SCA and ended with the acquisition of SCA-compliant SDR military radios.

Software-Defined Radios are embedded systems that process a very large quantity of data in real-time. As such, in addition to embedded GPPs, SDR platforms often use DSPs and FPGAs as well. Thanks to the SCA, software can be made very portable even for embedded GPPs and DSPs. SCA components are typically made of control source code and signal processing source code. Portability of SCA components can be affected when the signal processing part is optimized for special instructions sets such as the Streaming SIMD Extensions (SSE) for Pentium processors, the AltiVec instructions for PowerPC processors, or the NEON instructions for ARM processors.

Furthermore, portability is very limited when FPGA firmware is used for signal processing. Different FPGAs offer different resources. Often firmware is designed to use specific resources (e.g. block RAMs, FIFOs, DSP blocks, multipliers) that vary from one FPGA manufacturer to another. Besides, the FPGAs of a single manufacturer can vary significantly from one model to another in terms of such resources. As such, portability has been the holy grail of FPGA firmware designers. It is a research topic that has received a lot of attention over the years. Thus far, no one solution have prevailed over the others. Over time, the SCA has improved some aspects of portability for applications that use FPGAs. It did so by standardizing how software components running on DSPs and GPPs can interact with

components that run on FPGAs. With that approach, firmware can be adapted or rewritten for new FPGAs without having a serious impact on the software it interacts with. Nevertheless, the SCA does not improve the portability of the actual FPGA firmware.

One of the popular approaches to improve portability of high performance signal processing source code is to use domain-specific accelerators. The approach consists in writing source code for widely available libraries of domain-specific APIs that execute fast thanks to co-processors. Microsoft uses this approach with DirectX which offers a large number of functions that can be optimized to run on GPUs [1]. The same approach has also been used with FPGAs as co-processors [2, 3].

While the concept of accelerators can increase portability, different APIs must be used for different type of processing elements (GPPs, DSPs, GPUs, FPGAs). Relying on different APIs adds complexity for designers of applications for heterogeneous embedded systems. It also prevents portability across different processing elements.

Open Computing Language (OpenCL) is a framework for implementing software components that can execute across different processing elements [4]. It allows a developer to implement a function in source code that can be compiled for GPPs, DSPs, GPUs, and FPGAs. The following sections of this paper provide an overview of OpenCL, describe how SCA components can be built using OpenCL, and provide performance metrics. The paper concludes on how the SCA could be improved to offer better support for OpenCL.

2. THE OPEN COMPUTING LANGUAGE

OpenCL is an open and royalty-free standard maintained by a non-profit technology consortium called the Khronos Group [5]. It has been created to allow high-performance applications to execute on various devices of different architectures implemented by different vendors.

OpenCL greatly improves performances for a wide range of applications by allowing task-based and data-based parallel programming. With OpenCL, a computing system is made of a number of compute devices connected to a host processor. Compute devices are GPPs, GPUs, DSPs, or FPGAs. The host processor is a GPP.

An OpenCL application is made of two parts: kernels and a host program. OpenCL kernels are routines (algorithms) performing the data processing. Kernels are implemented in a C-like language and executed on the compute devices. A single compute device typically consists of many individual processing elements (PEs) and a kernel can run on all or many of the PEs in parallel. The host program runs on the host processor and is implemented using an application programming interface (API) to launch kernels on the compute devices and manage device memory. The OpenCL standard defines host APIs for C and C++; third-party APIs also exist for other programming languages [6, 7, 8]. An OpenCL framework consists of a library that implements the host APIs, and an OpenCL compiler for the target compute device(s).

2.1. Portability

The goal of OpenCL is to allow high-performance applications to run on any hardware. It provides portability by allowing the same source code to be compiled for different target compute devices. Host programs are compiled using the C/C++ compiler and the appropriate library for host APIs. Kernel programs can be pre-compiled for specific target compute devices before run-time. They can also be compiled on-the-fly at run-time for the required target devices.

OpenCL also extends C/C++ by providing standardized vector processing instructions and data types to exploit vector engines of the modern processors [9].

3. USING OPENCL TO INCREASE PORTABILITY OF SCA APPLICATIONS

SCA applications are made of one or many components that perform data processing. Every SCA component is made of configuration properties and ports to process data. Components also contain several implementations; one for each processing element it supports. For portable components, the different implementations are produced by building the same source code for the different processing elements.

For components that need to be optimized, the data processing source code needs to change significantly to exploit processor-specific instructions. The software part of a component that deals with control does not need to change much from one implementation to another.

However, using OpenCL, the data processing source code does not require any change to exploit the different processor architectures. In fact, OpenCL code can also be executed on FPGAs [10, 11]. OpenCL effectively reduces the development time required for a component to run on multiple processing elements including FPGAs. FPGA firmware is built using platform-specific features and requires very long development cycles.

3.1 A Simple Approach to using OpenCL with the SCA

SCA applications are deployed on SCA platforms via the execution of their components. The SCA Core Framework chooses an implementation for each component and executes it using an SCA device. The choice of the SCA device is made by matching the requirements of the component implementations with the capabilities being advertised by the SCA device. For instance, a component that only has one implementation that requires an x86 processor can only be executed by an SCA Device that advertises being capable of running x86 implementations.

Deploying a component implementation that uses OpenCL to perform signal processing works the same way as deploying a component that requires SSE or AltiVec instructions. The application component that is implemented using OpenCL simply needs to specify a requirement to be deployed on an SCA device that represents an OpenCL-capable processing element. Such an SCA device must advertise capabilities that identify its capability to host OpenCL programs.

4. CREATING AN OPENCL SCA COMPONENT

Figure 1 shows the structure of a typical SCA component where data to be processed is received via an input port and sent, after the processing is performed, to another component via an output port.

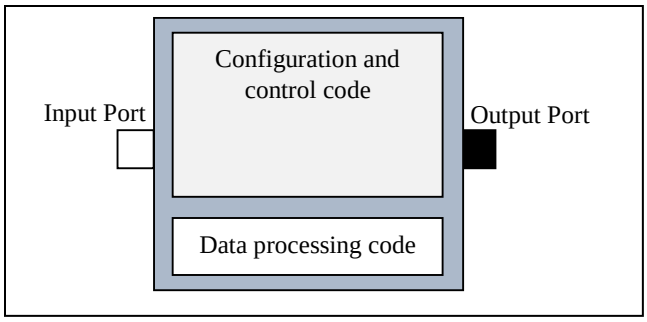

Figure 1. Structure of a typical SCA component.

The figure shows the distinction between the configuration and control code, and the data processing code. For an OpenCL SCA component, the host program is part of the configuration and control code, and the kernels are part of the data processing code. The kernels can potentially be executed on different compute devices (i.e. OpenCL-capable processing elements) when many compute devices are connected to the GPP where the host program runs. OpenCL provides APIs to list platform and compute device information, to obtain the identifiers for compute devices, and to specify which device should be used to execute kernels. One single SCA device can therefore load and execute kernels on any compute device connected to the GPP where the SCA device runs.

Typically, the source code for a kernel is located in a separate file from the OpenCL program source file. The OpenCL API offers several ways to create a program from which kernels are instantiated. A program can be created from a buffer containing program source code, from a buffer containing the program binaries, either in binary format specific to a device or in an intermediate representation that will be converted to the device-specific code format. The appropriate format is selected based on the level of portability and performance needed for an application.

For the SCA, this means the kernel files are not embedded in the source file for the SCA component implementation itself. The way to model this with the SCA is to define a software dependency between the SCA component implementation and the OpenCL kernel files it uses. Doing so will cause the SCA Core Framework to load the kernel files on the same SCA device used to execute the SCA component implementation.

4.1. Loading the kernels

Once an SCA component start running, it must load the kernels and instantiate them before the data processing starts. In our experiments, the kernel creation was done from during the initialization of the SCA application component (i.e. LifeCycle::initialize()). Kernel creation involves initializing OpenCL, listing and selecting compute devices, loading kernel files, and creating the kernels. This is all done using OpenCL APIs which makes calls to device drivers.

To be more portable, it is forbidden for SCA application components to make calls to native device drivers. However, just like applications are allowed to use several POSIX APIs, the SCA specification should allow OpenCL APIs since this standard is broadly supported across different types of processing elements. Alternatively, it would be possible to create an SCA-level API that SCA devices could implement for application components to use. This would prevent implementations of application component from being compiled and linked against native device drivers.

4.2. The Data Flow

OpenCL kernels use compute device memory to get input data and provide output data. The host program is responsible for creating compute device memory to be used by the kernels. The host is also responsible for copying data from its memory to the compute device memory and vice-versa if it is required.

SCA components usually receive and send data through ports. This means the data is in the memory of the host processor. Therefore, the input data received by an input port must be copied into the OpenCL compute device memory (H2D) before executing a kernel, and the output data produced by a kernel must be copied from the compute device memory to the host memory (D2H), after a kernel has executed. Figure 2 shows the data flow for every sequence of data being processed by an OpenCL SCA component.

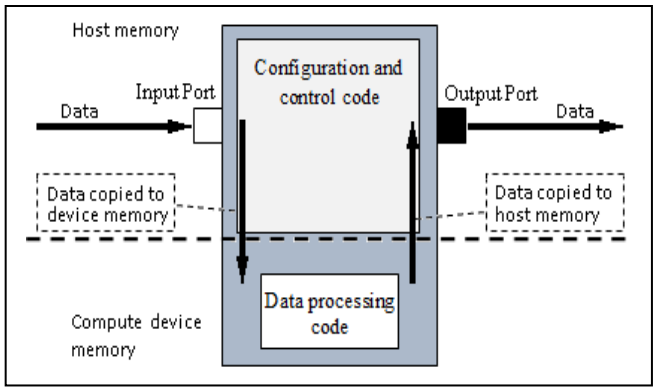

Figure 2. Data flow of data processed by an OpenCL SCA component.

Copying data between different memories affect the overall data processing performance. Copy of data can be avoided when the compute device is a CPU since the memory of the device is the same as the host. But, when the compute device is not a CPU then data must be copied. We have collected some metrics regarding this topic that will be presented in the next section.

5. METRICS

In this section, we discuss some metrics that can impact the performance of data processing using OpenCL. We also suggest solutions or research areas to address the issue we identify. To perform our experimentation, we used a desktop computer with an Intel i7-4770 CPU with 8 cores clocked at 3.40 GHz, 4GB of memory. We used the 64 bits version of Fedora 20 with the Linux kernel version 3.11.10-301. As for OpenCL, we used two compute devices. The first one was the CPU device of the Intel OpenCL platform with OpenCL 1.2. The second OpenCL device was PCI-E 3.0 NVIDIA GeForce GT 635 GPU using the NVIDIA OpenCL CUDA 7.0.41 platform with OpenCL1.1.

5.1 OpenCL Program Format

In section 2.1, we described that OpenCL brings portability by allowing the same source code to be compiled and executed for various compute devices with different hardware architecture. Building every kernel a head of time and packaging the binaries with the application components is in line with the common SCA. Each SCA application component contains several implementations of the component. Using OpenCL means each SCA component implementation will come with kernel binaries targeting a specific compute device. The deployment of an SCA application lead to the choosing of the right implementations of each component and each kernels based on the hardware available in the SCA platform.

However, with the proper driver support, kernels can be built on the fly at the moment the SCA application gets deployed. In such a case, the application is packaged with the kernels either in source code format or in an intermediate binary format which is portable across different compute devices. Indeed, OpenCL supports a format called Standard Portable Intermediate Representation (SPIR) for kernel binaries. SPIR is cross-platform and designed for heterogeneous parallel computing. It is based on LLVM IR [12].

Using this approach reduces the requirement for having several implementations of an SCA component and OpenCL kernels. If the SCA platform contains one GPP and several OpenCL compute devices, there is no need to prebuild all the kernels. The kernels can be built on the fly based on the selected compute devices. This approach also future-proves the SCA application since it supports any compute device that might be integrated in the future. In short, it makes the SCA application more portable to different SCA platforms that use the same GPP but different OpenCL compute devices. However, using this approach incurs a runtime cost during the deployment of applications since the OpenCL builder is invoked on the fly.

To evaluate the impact of selecting an approach over another, measurements have been made regarding the time it takes to create a kernel from source code, SPIR format, and from native binaries prebuilt for specific compute devices. The tests have been executed ten times for each file format and file size (i.e. small vs large) of the source code. To represent a small source file, we used a kernel routine implemented in 16 lines of code (LOC). We used a routine implemented with 398 LOC to represent a large source file. The SPIR binaries were created using the options "-x spir -spir-std=1.2" with the OpenCL compiler. Table 1 shows the average times it takes to create a kernel that is ready to be executed starting with above-mentioned 3 types of kernel files.

Table 1. Average time in µs to create a kernel based on source code file size.

	Small		Large	
Format in kernel file	**CPU**	**GPU**	**CPU**	**GPU**
Source code	13149	391	142089	447
Native binary	968	378	4381	396
Binary in SPIR	923	--	4187	--

As it can be seen from Table 1, creating a kernel from source code is surprisingly fast. Creating a kernel involves compiling and linking the kernel source code for different compute devices. For a CPU compute device, it takes approximately 13 to 142ms to create a kernel from source code. Doing the same for the GPU compute device only takes 0.3 to 0.5ms. Note that creating kernels only happens once each time an application is launched, no matter how long the application runs for. The reason it takes a different amount of time to create kernels for different compute devices is that different tool chains are used. Another surprising result is that creating a kernel for a GPU compute device takes about the same time whether from source code or from native binary. For a CPU compute device, creating a kernel from binary SPIR format takes about the same time as creating from native binary, even slightly faster. Since SPIR binaries are portable, this format represent the best solution for use with the SCA. The SPIR format also offers the side benefit of not exposing the kernel source code on the deployment platform.

5.2 Buffer Size

As mentioned before, the input data must be moved from the host memory to the target compute device memory on which a kernel will be executed. Similarly, the output data must be moved back to the host memory after the execution of the kernel. The time spent copying data affects the overall time required for OpenCL kernels to process data. Experiments have been conducted to measure the impact of copying of data across the bus that connects the host and the target devices.

The experiments used various buffer sizes, from 4KB for the size of small buffers to 3.125 MB for the size of large buffers (800 times the size of the small buffers). The measurements were averaged over twenty tests in each direction. Table 2 provides the averages in microseconds and illustrates the difference in performance between different types of compute devices. It also quantifies that the cumulative cost of copying data across memory types can be significant. Figure 3 shows the plotting of these numbers. NordiaSoft is currently investigating, with good success, different approaches to reduce the costs of moving data. Results to be published in a follow up paper.

Table 2. Average time to copy buffers.

Buffer size (KB)	**CPU**		**GPU**	
	H2D (µs)	**D2H (µs)**	**H2D (µs)**	**D2H (µs)**
4	5	9	10	12
32	7	12	19	19
320	32	42	101	104
640	67	75	191	312
960	112	105	406	464
1280	155	153	468	614
1600	193	161	520	694
1920	247	186	577	814
2240	274	209	653	903
2560	333	234	706	1020
2880	608	296	746	1194
3200	694	372	794	1307

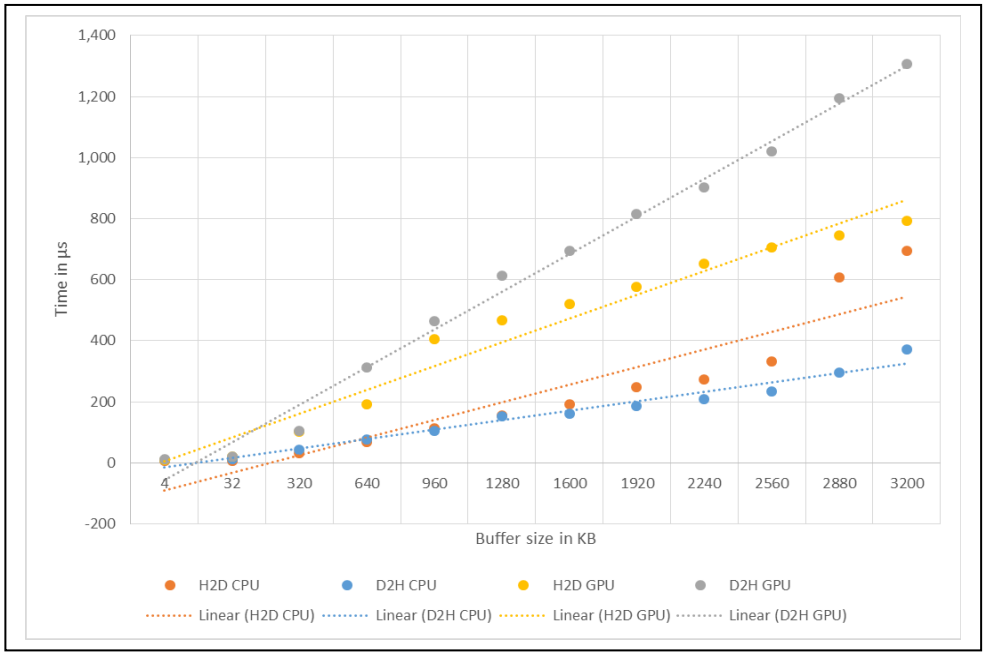

Figure 3. Average time to copy buffers from H2D and D2H.

6. CONCLUSION

OpenCL is effective to increase the portability of SCA applications across heterogeneous platforms. It allows application components to be portable between GPPs, DSPs, GPUs, and FPGAs. In short, OpenCL addresses directly the number one innovation from the top 10 most wanted innovations as defined the Wireless Innovation Forum. The paper describes how the SCA can benefit from OpenCL. It explains how OpenCL SCA components can support multiple compute devices with a single implementation of the signal processing source code.

The paper underlined the fact that portability for signal processing functions can be achieved at the source code level and at the binary level which offers more protection for intellectual property. Metrics have been presented to illustrate how fast it is to instantiate OpenCL kernels. The paper also provided metrics that show the performances associated with moving data across different types of memory.

A simple approach to support OpenCL with SCA has been presented. It described how an SCA Device must advertise its capabilities to execute OpenCL kernels. It also explained how SCA application components can integrate OpenCL kernels. We have identified some areas of potential improvement for the SCA specification to better support OpenCL.

Finally, the paper showed how the copy of data between the OpenCL host processor and a target compute device can potentially affect real-time performances. More research can be performed on this topic to alleviate the issue.

7. REFERENCES

[1] F. D. Luna, Introduction to 3D Game Programming with DirectX 10, WordWare Publishing Inc., Sudbury, MA, USA, 2008.

[2] W. Zhang, V. Betz, and J. Rose, Portable and Scalable FPGA-Based Acceleration of a Direct Linear System Solver, ACM Transactions on Reconfigurable Technology and Systems, Vol. 5, No. 1, Article 6, March 2012.

[3] G. C. T. Chow, K. Eguro, W. Luk, and P. Leong, A Karatsuba-based Montgomery Multiplier. FPL '10 Proceedings of the 2010 International Conference on Field Programmable Logic and Applications. 2010.

[4] http://en.wikipedia.org/wiki/OpenCL.

[5] The Khronos OpenCL Working Group, The OpenCL Specification version 2.0, 2014, https://www.khronos.org/opencl/.

[6] http://mathema.tician.de/software/pyopencl/

[7] https://code.google.com/p/javacl/

[8] https://github.com/Nanosim-LIG/opencl-ruby

[9] M. Scarpino, *OpenCL in Action*, Manning Publications Co., Shelter Island, 2012.

[10] R. Brueckner, How OpenCL Could Open the Gates for FPGAs, 2015, http://insidehpc.com/2015/02/how-opencl-could-open-the-gates-for-fpgas/.

[11] Implementing FPGA Design with the OpenCL Standard, November 2013, https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01173-opencl.pdf.

[12] The Khronos Group Inc., The SPIR™ Specification version 1.2, 2014, https://www.khronos.org/registry/spir/