Using OpenCL to Increase SCA Application Portability

Steve Bernier (NordiaSoft, Gatineau, Qc, Canada; Steve.Bernier@NordiaSoft.com);
François Lévesque (NordiaSoft, Gatineau, Qc, Canada; Francois.Levesque@NordiaSoft.com);
Martin Phisel (NordiaSoft, Gatineau, Qc, Canada; Martin.Phisel@NordiaSoft.com);
Dmitry Zvernik (NordiaSoft, Gatineau, Qc, Canada; Dmitry.Zvernik@NordiaSoft.com);
David Hagood (Cobham, Wichita, KS, USA; David.Hagood@Cobham.com);

ABSTRACT

The Software Communications Architecture has become the de facto standard to build Software Defined Radio radios. Over one hundred thousand SCA military radios have been deployed worldwide by several nations. The SCA offers a component-based operating environment for the creation of portable applications. SCA applications are portable across different heterogeneous embedded distributed system.

For performance reasons, application software gets optimized using specialized instructions sets supported by General Purpose Processors, Digital Signal Processors Graphical Processing Units. As a result, the level of portability of the source code can decrease significantly. Moreover, portability is considered to be a major challenge when Field Programmable Gate Arrays are used.

Specialized instruction sets are widely used for high performance military radio platforms. Consequently, finding a solution to increase portability of SCA applications across different operating environments could provide significant cost reductions. This paper describes how the Open Computing Language (OpenCL™) can be used in conjunction with the SCA to increase the portability of applications that need to perform intensive signal processing.

1. INTRODUCTION

The Software Communications Architecture (SCA) was created to standardize the software architecture of real-time embedded systems. The SCA is mainly used to build Software-Defined Radios (SDRs). The standard was created for the Joint Tactical Radio System (JTRS) program, a US DoD program that funded the development of military SDRs. The JTRS program started by funding the definition of the SCA and concluded with the acquisition of SCA-compliant SDR military radios. The main goal of the JTRS program was to allow applications that implement communications standards, called waveform applications, to easily be ported from one radio platform to another.

The SCA is a software architecture that makes applications very portable across different heterogeneous systems. The SCA standardizes how applications are launched and how they interact with hardware. It also defines how applications are packaged, installed, deployed, and controlled. In its current state, the SCA standard allows manufacturers to reach a high level of portability for applications that have been implemented for General Purpose Processors (GPPs) and Digital Signal Processors (DSPs). SCA components can easily be ported to various operating environments. An Operating environment is defined as being made of an operating system, a specific processor, and a one-to-many communication buses. The SCA even makes the GPP/DSP software that interacts with a Field Programmable Gate Array (FPGA) more portable.

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. Implementing optimized source code for DSPs and FPGAs is well-known to be challenging and time-consuming. It also reduces the level of portability of source code [1]. It should therefore come as no surprise that application portability is the number one innovation on the top ten list of the most wanted wireless innovations as compiled by the Wireless Innovation Forum (WInnF[1]).

OpenCL is a standard for writing signal processing intensive applications that are portable. OpenCL was initially designed by Apple and it is now widely embraced by major chip manufacturers such as Intel, AMD, IBM, QualComm, Samsung, and NVIDIA [2]. The standard is open and royalty-free. It is maintained by a non-profit technology consortium called the Khronos Group [3].

OpenCL allows a developer to implement a signal processing function in source code that can be cross-

[1] www.wirelessinnovation.org

compiled for GPPs, DSPs, GPUs, FPGAs and other specialized processors or hardware accelerators. Combining the OpenCL with the SCA holds the promise of making SCA applications much more portable across different platforms. The remainder of this paper explains how OpenCL can be used with the SCA. Section 2 describes the structure of SCA applications. Section 3 provides a short introduction to OpenCL. Section 4 describes how SCA components can be built using OpenCL while section 5 provides performance metrics. The paper concludes on how the SCA could be improved to offer better support for OpenCL.

2. SCA APPLICATION PORTABILITY

SCA applications are made of several software components interconnected with each other. For instance, the SCA FM Transmitter waveform application shown in figure 1 is made of 3 components: a voice filter, a squelch injector, and a FM modulator. The application is also connected to an audio device component and a RF device component. The audio device component represents a sound card that provides microphone voice samples. The RF device represents a transceiver which transmits modulated voice signals over the air.

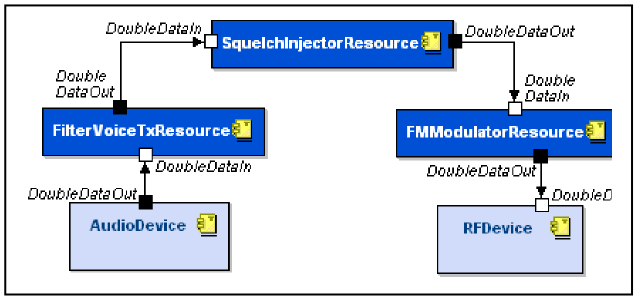

Figure 1. SCA FM Transmitter waveform application

Much like a chip in electronics, a SCA software component performs a simple task that can be used in many applications. Each software component has a number of ports to which other components can connect to exchange data or perform control.

SCA components are made of metadata and a number of binaries. The metadata depicts the characteristics of a component. It describes each property and port of the component in details. It also defines for which operating environment this component has been implemented. For instance, assume a SCA Frequency Modulator component can run on the following three operating environments: PPC/VxWorks, ARM/INTEGRITY, and x86/Linux. Such a component would come with three executable binary files; One for each operating environment. With the SCA, each of the three executable binaries is called a component "implementation".

Launching a SCA component involves choosing an implementation that matches the characteristics of the desired target operating environment. The SCA Core Framework is responsible for picking the appropriate implementation of a component based on the capabilities of the hardware platform. The launching process results in executing the binary file associated with the selected implementation. The SCA mandates that every implementation of a SCA component offer the same behavior, ports, and properties. Thus, once a component is launched, it will behave the same way no matter which implementation has been selected. The concept of implementations is how the SCA guaranties components are portable. An SCA application is only portable if its components contain more than one implementation each. Consequently, the level of portability of an SCA application is proportional to the number of implementations its components contain.

Porting a component to a new operating environment involves adding a new implementation to the component and creating a new executable binary file. The binary file is produced from the appropriate source code for the new operating environment. Nothing in the SCA requires the new binary file to be produced from the same source code as for the existing implementation binaries. Application portability means the application can run, unchanged, on many different platforms which is the case even when all the implementations of its components are produced from different source code. Ideally, to reduce cost and time-to-market, it would be better to use the same source code to address different operating environments. However, in the case a SCA component needs to support very different operating environments, it is perfectly valid to use different source code. Of course, the source code can be produced by any tool chain or manually.

The source code for each implementation of an SCA component is made of two main pieces: control source code and signal processing source code. The control source code takes care of routing input data to the proper signal processing functions. It also takes care of routing the processed data to other components. The control source code handles a number of additional things including what happens when a component is started, stopped, connected, disconnected, and more. The SCA operating environment rules make the control source code very portable.

As for the signal processing source code, the second piece of an SCA component, it is responsible for handling the data. It transforms input data and produces output data.

The transformation of data can be influenced by a number of properties the component offers. For instance, changing the value of a property called "Code Rate" could change the behavior of the signal processing source code located inside a Vocoder component.

Signal processing can greatly benefit in performances from the use of special accelerators such as SSE with Intel processors, AltiVec with PowerPC processors, NEON with ARM processors, MAPLE-B with Freescale processors, and more. Even though each of the above accelerators are designed to accelerate signal processing, they differ from one another. It is therefore difficult to make signal processing source code portable across different accelerators.

Furthermore, the use of FPGAs to implement signal processing is very common with Software Defined Radios. However, FPGA software (called firmware) is notorious for not being portable. Firmware is designed to use specific resources (e.g. block RAMs, FIFOs, DSP blocks, multipliers) that vary from one FPGA manufacturer to another. In fact, FPGAs can vary significantly within the same family of products from the same manufacturer. Portability of FPGA firmware is a research topic that has received a lot of attention over the years. Many solutions have been proposed [4, 5, 6, 7, 8], but none have gained broad acceptance.

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 signal processing algorithms that run on FPGAs [9, 10]. With such an approach, firmware can be adapted or rewritten for new FPGAs without having a serious impact on the software that runs on GPPs or DSPs. Nevertheless, the SCA does not help the signal processing source code become more portable. Instead, the SCA concentrates on the portability of SCA components and applications.

Improving the portability of source code is the holy grail of the signal processing industry. One of the popular approaches to improve portability of signal processing source code is to use libraries that abstract domain-specific accelerators. Microsoft uses this approach with DirectX which offers a large number of functions that can be optimized to run on GPPs and GPUs [11]. Different frameworks have also been used with FPGAs [12, 13]. However, most approaches are limited to specific types of accelerators or to specific operating systems.

The most recent attempt to solve the problem of source code portability across different types of accelerators is OpenCL. This approach is very promising since it has been adopted by a large number of companies that manufacture different types of accelerators.

3. THE OPEN COMPUTING LANGUAGE

The Open Computing Language (OpenCL™) offers the possibility of implementing signal processing software that can execute across different types of accelerators. OpenCL has been created to allow high-performance signal processing source code to execute on GPPs, DSPs, GPUs, FPGAs and other specialized processors or hardware accelerators. OpenCL allows a developer to implement source code that can be cross-compiled for different types of accelerators. It can be used for a wide range of task-based and data-based parallel programming.

The standard defines a programming language that is largely based on the C language (C99) and adds a number of built-in functions for scalar and vector operations [14]. The language also allows source code to handle both the host memory and the accelerator memory. Third-party mappings exist for different programming languages [15, 16, 17]. OpenCL also provides APIs for a host program to select and control any accelerator, called a compute device in OpenCL. The APIs are used by host programs to run signal processing functions, called kernels, on a compute device.

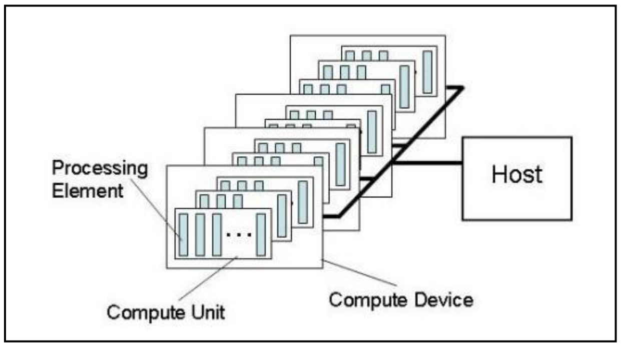

Figure 2. OpenCL platform model

With OpenCL, a system is viewed as being made of a number of compute devices (e.g. GPPs, GPUs, DSPs, FPGAs) connected to a host processor (i.e. GPP). A single compute device can be made of several compute units each of which contain multiple processing elements (Figure 2). The execution of a single kernel can run on all or many of the processing elements in parallel. How a compute device is subdivided into compute units and processing elements is defined by the hardware manufacturer as it adds support for OpenCL.

OpenCL provides portability by allowing the same source code to be cross-compiled for different compute devices. Host programs are compiled using the C/C++ compiler and the appropriate library for host APIs. The kernels need to be compiled and linked using the tools provided by the manufacturer of the accelerator. Those tools typically come with the OpenCL drivers. A kernel must be compiled for each accelerator it needs to be executed on. The compilation typically happens before run-time. However, in some cases, the kernels can also be compiled programmatically during run-time. As indicated before, no changes are needed to the source code of the kernels to be compiled for various accelerators.

One of the first things a host program does is use OpenCL APIs to discover which compute devices are available. Compute devices become available to a GPP processor by installing the required OpenCL device drivers. Intel provides OpenCL drivers that exploit the SSE and AVX accelerators in most Intel processors. OpenCL drivers also exist for many GPUs. Most System-on-Chip (SoC) solutions combine a number of GPP and GPU cores which are supported by OpenCL. In fact, OpenCL drivers even exist for a number of DSP [18] and FPGA processors [19, 20, 21].

When the host program gets access to more than one compute device, it needs to decide which to use to run the kernels. It is important to remember that kernels are contained in separate files from the host program. The host program can execute kernels by getting access to the binaries created from the source code of the kernels. The host program needs to create a command queue to execute kernels on a specific compute device. It can create more than one command queue to execute kernels on different compute devices. OpenCL offers a rich API to allow the host program to schedule the execution of kernels in a specific order. The host program can express very sophisticated dependencies that exist between different kernels.

Before the execution of a kernel can start, the host program must handle all the input buffers required by the kernel. The buffers must be copied from the host memory to the compute device memory where the kernel will execute. In the case the host program targets a local accelerator, OpenCL drivers don't make unnecessary memory copies. For instance, if the host program runs on an Intel processor and targets the SSE/AVX accelerators, the OpenCL driver does not make copies of the buffers. The Intel OpenCL driver for the Intel processor knows the host processor and the accelerators have access to the same memory. But in the case the host program runs on an Intel processor and targets a GPU accelerator, the GPU OpenCL driver will copy the input buffers from host memory to the compute device memory.

When the host program schedules several kernels for execution on the same compute device, it is possible for the output data of one kernel to become the input of the next kernel. In such a case, the data remains in the compute device memory. The host program does not need to copy the data back and forth between the host and compute device memory. After the execution of the last scheduled kernel, the host program usually copies the output data of the kernel from the compute device back into the host memory. If the host program needs to re-execute the kernels, it must reschedule the execution and handle the input and output buffers accordingly. This is very common with SDR applications since they need to process the data that continuously flows through the radio.

4. USING OPENCL TO INCREASE PORTABILITY OF SCA APPLICATIONS

As stated before, the source code that implements the control part of the component does not need to change much from one implementation to another. However, in many cases, the source code that performs signal processing needs to be optimized for specific accelerators. This is where OpenCL can help make SCA applications more portable.

Using OpenCL, a developer can create several component implementations without having to craft different source code for the signal processing. OpenCL can be used to reduce the development time required for a component to support multiple types of accelerators, including FPGAs. Since the signal processing source code does not need to change, using OpenCL allows SCA applications to quickly benefit from new and more powerful accelerators as they get released.

Creating a SCA component implementation using OpenCL requires that the implementation source code plays the role of the host program. The component implementation must initialize the OpenCL environment and select a compute device in order to create the appropriate command queues. It must also take care of scheduling the execution of the kernels it needs. The component implementation, as usual, must be compiled for a specific host processor. The kernels are however compiled using the appropriate OpenCL compiler. In short, a component implementation is made of a host program which is distinct from its kernels.

4.1. Loading the kernels

Launching the execution of a component implementation that uses OpenCL requires the SCA Core Framework selects a SCA Device that meets the requirements of both the component implementation and the kernels. For example, launching an x86/Linux component implementation that comes with OpenCL kernels requires a SCA Device that controls an x86/Linux/OpenCL operating environment.

Since, the kernel binaries are located in separate files from the component implementation, the authors propose to model the component implementation by defining a software dependency between the implementation and the OpenCL kernel files it needs. Also, the component implementation will need to define its usual requirements while the kernels need to declare a requirement for a specific OpenCL compute device. With such an approach, any standard SCA Core Framework will be able to load the kernel files on the same SCA Device as the one used to execute the SCA component implementation.

Once the component implementation gets launched, it will be able to schedule the execution of its kernels by getting access to them via the file system. In the experiments conducted for this paper, the kernel creation was done during the initialization phase of the SCA application component. Kernel creation involves initializing OpenCL, listing and selecting compute devices, loading kernel files, and instantiating the kernels. This is all done using OpenCL APIs which makes calls to device drivers.

To maximize portability, it is forbidden for SCA application components to make calls to native device drivers. Therefore, making calls to the OpenCL drivers represents a problem. However, the authors believe the SCA specification could allow component implementation to use OpenCL APIs since the standard is broadly supported across different types of processing elements. This approach would be in line with the current approach to allow the use of CORBA and POSIX. A new Application Environment Profile (AEP) standard document would be required. Alternatively, it would be possible to create an API that SCA devices could implement for application components to use. This could theoretically prevent application component implementations from being compiled and linked directly against OpenCL drivers.

4.2. The Data Flow

Figure 3 shows the distinction between the control source code and the signal processing code. For an OpenCL SCA component, the host program is part of the control source code, and the kernels represent the signal processing part. As stated before, the kernels can be executed on different compute devices. OpenCL kernels use compute device memory to get input data and to provide output data. The host program is responsible for creating compute device memory to be used by the kernels. The host program is also responsible for copying data from its host memory to the compute device memory and vice-versa.

SCA components usually receive and send data through ports. This means the data is located in the memory of the host processor. Therefore, the implementation of an OpenCL-capable component needs to copy its input data into the OpenCL compute device memory (H2D) before executing a kernel. Likewise, the component implementation needs to copy the output data produced by a kernel from the compute device memory to the host memory (D2H). Figure 3 shows the data flows through an OpenCL SCA component.

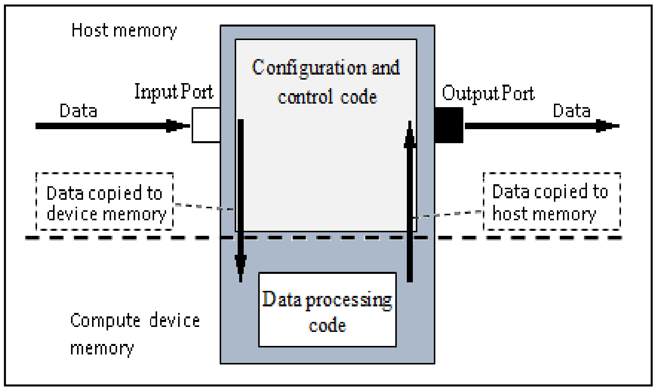
Host memory
Configuration and control code
Input Port
Output Port
Data
Data
Data copied to device memory
Data copied to host memory
Compute device memory
Data processing code

Figure 3. Data flow of data processed by an OpenCL SCA component.

5. METRICS

Copying data between different memories affects the overall data processing performance. This section provides some metrics to help measure the impact of copying data across the different types of memory. Our experimentation was performed in two groups. The first group of experiments was performed with OpenCL on an Intel SSE/AVX engine and on a NVIDIA GPU. The second group of experiments was done using an Altera Cyclone V SoC.

5.1. OpenCL for GPP and GPU

Our experiments were conducted on a desktop computer with an Intel i7-4770 CPU with 4 hyper-threaded cores clocked at 3.40 GHz with 4GB of DDR3 memory. We used the 64-bit version of Fedora 20 with the Linux kernel version 3.11.10-301. As for OpenCL, we used two compute devices. The first compute device was the Intel processor running the drivers that come with the Intel® SDK for OpenCL™ Applications. These drivers offer OpenCL version 1.2 which exploits the SSE and AVX instructions. The second OpenCL device was the GPU of the PCI-E 3.0 NVIDIA GeForce GT 635 graphics board running the drivers that come as part of the NVIDIA OpenCL CUDA 7.0.41 platform with OpenCL version 1.1.

5.1.1. OpenCL Program Format

Section 3 describes 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 and packaging the binaries with the application components is in line with common SCA practices. In such a case, each SCA application component would contain 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 leads 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 [22].

Using on-the-fly compilation eliminates the need for pre-compiling all the OpenCL kernels and as a result, it reduces the number of individual component implementations that are required. For example, assume a platform provides an ARM processor that has access to both a GPU and the ARM NEON accelerator. Also assume there is an application that is made of two SCA components, each having its own OpenCL kernel. Using the traditional approach with binary kernels, one kernel would need to be compiled for OpenCL/GPU while the other would need to be compiled for OpenCL/NEON. If all the flexibility is needed, both kernels would have to be compiled for both OpenCL compute devices. This would require that each SCA component be made of two implementations: one implementation for the ARM with a dependency to the OpenCL/GPU kernel binary and a second implementation for the ARM with a dependency to the OpenCL/NEON kernel binary. However, using the kernel source code instead would only require one component implementation for the ARM with a dependency to the source code file (or SPIR file). As the kernel source code would get loaded into the ARM (NEON accelerator) or the GPU, the OpenCL driver would dynamically compile the source code.

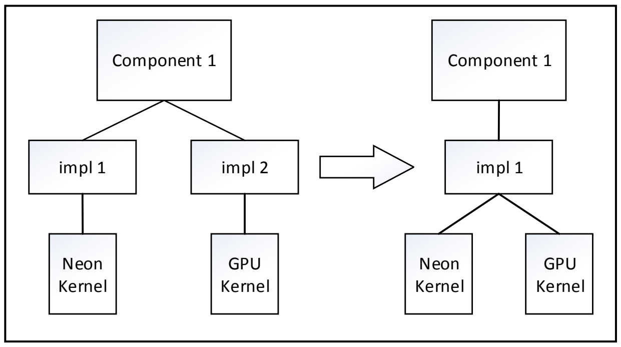

Figure 4. Data flow of data processed by an OpenCL SCA component.

Using the source code approach enables the SCA application to be future-proof. Such an SCA application can support any compute device that might be released in the future for as long as it comes with the ability to compile on-the-fly (Figure 4). It also 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

	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	--

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

To evaluate the impact of an approach over the other, measurements have been made regarding the time it takes to instantiate 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, a kernel implemented in 16 lines of code (LOC) was used. Another kernel implemented with 398 LOC was used 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 instantiate a kernel that is ready to be executed starting with above-mentioned 3 types of kernel files.

As it can be seen from Table 1, instantiating a kernel from source code is surprisingly fast. Kernel instantiation involves compiling and linking the kernel source code for different compute devices. For a CPU compute device, it takes approximately 13 to 142ms to instantiate a kernel from source code. Doing the same for the GPU compute device only takes 0.3 to 0.5ms. Note that instantiating 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 instantiate kernels for different compute devices is that different tool chains are used. Another surprising result is that instantiating 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, instantiating a kernel from binary SPIR format takes about the same time as from native binary, even slightly faster. Since SPIR binaries are portable, this format represents 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.

5.1.2. 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 produced by a kernel must be moved back to the host memory. 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 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 5 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.

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

Table 2. Average time to copy buffers.

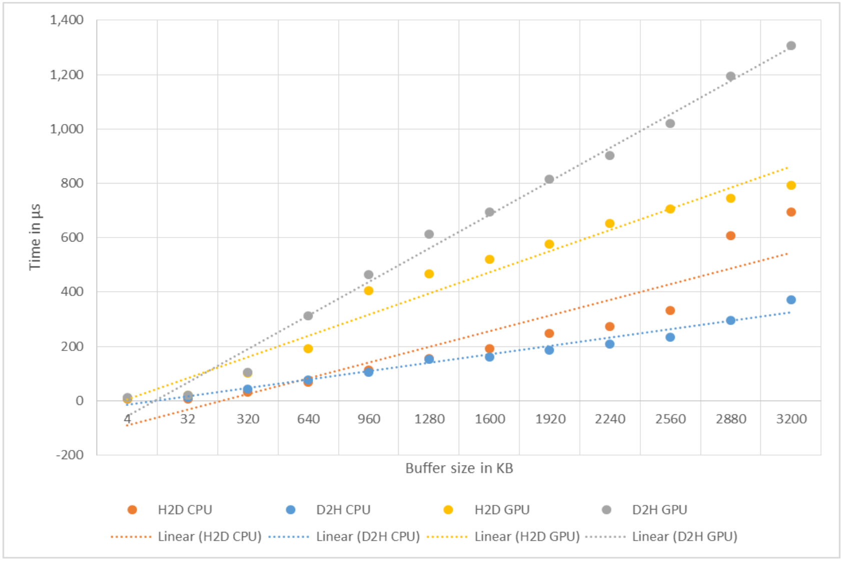

Figure 5. Average time to copy buffers from H2D and D2H for CPU/ GPU.

Traditional GPU cards are made to render sophisticated graphics based on massive amounts of data coming from the GPP [23]. Table 2 clearly illustrates the predisposition of the GPU to more efficiently copy data from the host to the device than the other way around. The table also shows a sharp increase in time for copying more than 2880 KB which is believed to be related to memory caching. Above a certain threshold, data can only be held in part by the low-level memory cache.

5.2. OpenCL for Altera Cyclone V SoC

The previous experiments were performed with OpenCL on an Intel SSE/AVX engine and on a NVIDIA GPU. The following group of experiments was done using an Altera Cyclone V Terasic SoCKit board [24]. This development kit is one of the boards Altera recommends for OpenCL development. The kit's hardware design platform is built around the Altera Cyclone V System-on-Chip (SoC) FPGA[2], which combines a Dual-Core ARM Cortex™-A9 MPCore™ processor clocked at 800MHz with 512KB of shared L2 cache and 64KB of scratch RAM, programmable logic with 110K logic elements (LEs). Altera's SoC integrates an ARM-based hard processor system (HPS) consisting of processor, peripherals and memory interfaces tied with the FPGA fabric using an interconnect backbone. The board contains 2GB of DDR3L memory separated in halves between the HPS and the FPGA.

The ARM Linux BSP and Linux kernel image are supported by the Altera software developer community RocketBoards.org. User space and runtime OpenCL libraries are supplied by Altera as part of the AOCL SDK v14.0. The OpenCL kernels are built for Cyclone V FPGA by using Quartus 14.0 [25].

5.2.1. OpenCL Program Format

In order to build OpenCL kernels for a FPGA, the kernels must be compiled with the OpenCL FPGA compiler that usually comes from the FPGA vendor. The compiler parses the source code of the kernels and performs some performance optimizations. In addition, it identifies which Intellectual Property (IP) Cores will be needed. Finally, it performs the place-and-route step in order to create a final FPGA image containing the OpenCL kernels. The image also contains necessary infrastructure like blocks for kernel memory clocks, host interface controller, kernel interface controller, and more [26, 27].

It should be mentioned that it is not possible to dynamically compile individual OpenCL kernels for a FPGA. All the kernels must be combined into a single FPGA image [26, 27, 28, 29]. This differs from the case of GPP and GPU accelerators that can be used to dynamically load new kernels during runtime. From an SCA standpoint, this difference is important. With a single image that contains all the kernels, the SCA components cannot load their individual kernels dynamically into the FPGA. Nevertheless, each SCA component can still instantiate the kernels contained in the FPGA image individually. However, the instantiation is done using the FPGA image, which means every SCA component must have access to it. In SCA, this can be accomplished by using software dependency to the FPGA image file.

5.1.2. Buffer size

Buffer size (KB)	FPGA	
	H2D (µs)	D2H (µs)
4	65	49
32	327	235
320	2961	2438
640	5858	4879
960	8836	7342
1280	11709	9790
1600	14623	12261
1920	17477	14753
2240	20463	17204
2560	23266	19654
2880	26166	22112
3200	28984	24587

Table 3. Average time to copy buffer to/from FPGA

The case of OpenCL/FPGA is no different than any other OpenCL accelerator, data buffers must be copied across the bus that connects the host and target processors. Table 3 presents average measurements that represent the time it takes to copy buffers of various sizes from host memory to compute device memory (H2D) and vice versa (D2H). The measurements are in microseconds and represent the average of ten experiments conducted for each buffer size in each direction. All the experiments are conducted using the Altera Cyclone V Terasic SoCKit board [24]. To be more specific, Table 3 shows delays associated with copying buffers from an ARM processor to an FPGA and vice versa. Figure 6 shows the plotting of the measurements provided in Table 3.

[2] Altera part number 5CSXFC6D6F31C6N

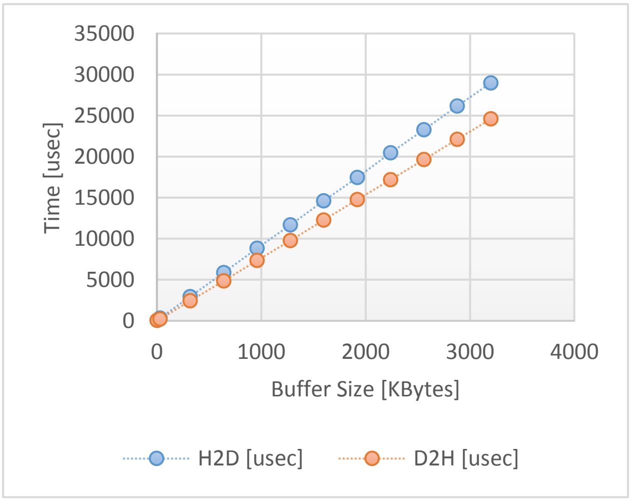

Figure 6 - Average time to copy buffers from H2D and D2H for CPU/FPGA.

The experiments conducted on the Altera Cyclone V Terasic SoCKit board (measurements listed Table 3) show data does not move as fast as with the experiments conducted on the desktop computer (measurements listed Table 2). This can be explained by the fact that the Altera Cyclone V Terasic SoCKit board is significantly slower than the hardware in the desktop computer. Altera Cyclone V contains an ARM processor that has 2 cores clocked at 800MHz whereas the desktop i7 processor has 4 hyper-threaded cores clocked at 3.4 GHz and GeForce GTX 950M GPU cores are clocked at 993MHz each. The speed of the memory bus is also different between the two experiments. The SoCKit board has a 2GB of DDR3L clocked at 400MHz whereas the i7 processor has 4GB of DDR3 memory clocked at 1600 MHz, and 4GB of DDR3 memory tied to GeForce GTX 950M GPU clocked at 1800MHz.

6. CONCLUSION

This paper describes how OpenCL, combined with the SCA, can be used to address the number one innovation of the top 10 most wanted innovations as defined by the Wireless Innovation Forum.

The experiments performed for this paper clearly demonstrate that OpenCL kernels are fully portable across GPPs, GPUs, and FPGAs. During the experiments, not a single line of source code was changed in the kernels. The paper also describes how OpenCL kernels can be combined with SCA components for ultimate level of portability across heterogeneous embedded distributed systems. Furthermore, according to publicly available documentation [30], the authors of this paper have all the reasons to believe OpenCL would be as portable for DSPs.

The paper underlines 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 in the case of source compilation. The paper also provides metrics that show the performances associated with moving data across different types of memory.

A simple approach to support OpenCL with SCA is presented. It describes how an SCA Device must advertise its capabilities to execute OpenCL kernels. It also explains how SCA application components can integrate OpenCL kernels. It should be noted that since every version of the SCA specification allows binaries to be loaded and executed, the approach proposed in this paper to integrate OpenCL applies to all versions of the SCA up to the latest version (i.e. version 4.1). The paper identifies some areas of potential improvement for the SCA specification to better support OpenCL.

Finally, the paper shows 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 explore optimization possibilities.

7. REFERENCES

[1] C. Baldwin, E. Mohsen, ASIC or FPGA: Why Not Plan For Portability?, Chip Design Tools Technologies & Methodologies

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

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

[4] J.L. Tripp, P.A. Jackson, B. L. Hutchings, Sea Cucumber: A Synthesizing Compiler for FPGAs, Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream, Volume 2438 of the series Lecture Notes in Computer Science, pp 875-885, August 2002, Springer-Verlag

[5] Z. Guo, B. Buyukkurt, J. Cortes, A. Mitra, W. Najjar, A Compiler Intermediate Representation for Reconfigurable Fabrics, International Journal of Parallel Programming October 2008, Volume 36, Issue 5, pp 493-520, Springer-Verlag

[6] E.S. Chung, J.C. Hoe, K. Mai, CoRAM: an in-fabric memory architecture for FPGA-based computing, Proceeding FPGA '11 Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays, pp 97-106, February 2011, ACM

[7] R. Kirchgessner, G. Stitt, A. George, H. Lam, RC: A virtual FPGA platform for applications and tools portability, Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream, Conference: Proceedings of the ACM/SIGDA 20th International Symposium on Field

Programmable Gate Arrays, FPGA 2012, Monterey, California, USA, February 22-24, 2012
[8] B. Klöpper, N. Cranston, M. Aleksy, M. Dix, Developing portable FPGA applications - A literature review, Industrial Informatics (INDIN), 2013 11th IEEE International Conference, 29-31 July 2013, pp 123-128, IEEE
[9] Joint Tactical Networking Center Standard Modem Hardware Abstraction Layer Application Program Interface, Version 3.0.0, 02 Oct 2013, JTNC. http://www.public.navy.mil/jtnc/sca/Documents/SCA_APIs/API_3.0_20131002_Mhal_withErrata.pdf
[10] Joint Tactical Radio System Standard MHAL on Chip Bus Application Program Interface, Version 1.1.5, 26 June 2013, JTNC. http://www.public.navy.mil/jtnc/sca/Documents/SCA_APIs/API_1.1.5_20130626_Mocb.pdf
[11] F. D. Luna, Introduction to 3D Game Programming with DirectX 10, WordWare Publishing Inc., Sudbury, MA, USA, 2008.
[12] 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.
[13] 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.
[14] M. Scarpino, *OpenCL in Action*, Manning Publications Co., Shelter Island, 2012.
[15] OpenCL mapping for Python. http://mathema.tician.de/software/pyopencl/
[16] OpenCL mapping for Java. https://code.google.com/p/javacl/
[17] OpenCL mapping for Ruby. https://github.com/Nanosim-LIG/opencl-ruby
[18] TI implementation of the Khronos OpenCL 1.1 specification http://downloads.ti.com/mctools/esd/docs/opencl
[19] R. Brueckner, How OpenCL Could Open the Gates for FPGAs, 2015, http://insidehpc.com/2015/02/how-opencl-could-open-the-gates-for-fpgas/.
[20] 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.
[21] M. Parker, M. Jarvis, The Most Under-rated FPGA Design Tool Ever, EE Times, 2015.
[22] The Khronos Group Inc., The SPIR™ Specification version 1.2, 2014, https://www.khronos.org/registry/spir/
[23] Y. Fuji, T. Azumi, N. Nishio, S. Kato, M. Edahiro, Data Transfer Matters for GPU Computing,
[24] SoCKit - the Development Kit for New SoC Device http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=167&No=816
[25] Altera OpenCL for Arrow SoCkit Setup Instructions v.14.0.01, 11/25/2014, http://rocketboards.org/foswiki/pub/Documentation/ArrowSoCKitOpenCL/SoCkit_OpenCL_Setup-v14.0--2014-11-25.pdf?t=1457556033
[26] Altera SDK for OpenCL Programming Guide (UG-OCL002) 2015.11.02, https://www.altera.com/opencl
[27] Altera RTE for OpenCL Getting Started Guide (UG-OCL005) 2015.11.02, https://www.altera.com/opencl
[28] Altera SDK for OpenCL Best Practices Guide (UG-OCL005) 2015.11.02, https://www.altera.com/opencl
[29] Altera SDK for OpenCL Custom Platform Toolkit User Guide (UG-OCL007) 2015.11.02, https://www.altera.com/opencl
[30] Texas Instrument OpenCL™ Runtime Documentation, http://downloads.ti.com/mctools/esd/docs/opencl/index.htm