INCREASING PERFORMANCES OF SCA APPLICATIONS THAT USE OPENCL

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 (Cobham, Wichita, Kansas, USA; David.Hagood@Cobham.com);

ABSTRACT

The Open Computing Language (OpenCL^{TM}) can be used in conjunction with the Software Communications Architecture (SCA) to build very portable applications that execute across heterogeneous platforms consisting of General Purpose Processors (GPPs), Digital Signal Processors (DSPs), Field Programming Gate Arrays (FPGAs), and Graphics Processing Units (GPUs). This paper starts with an overview of how SCA components can be built using OpenCL. It compares performance metrics of an application implemented as several OpenCL-SCA components with the metrics of a variation of the application that uses a reduced-copy technique for the data being processed. The paper provides a detailed discussion on how the data flows in and out of the OpenCL device memory as it travels from a component to the next. The paper describes a novel approach that minimizes the number of copies made as the data flows through the different signal processing components of an SCA application. The paper concludes by identifying further research topics that could be investigated on this subject.

1. INTRODUCTION

The Software Communications Architecture (SCA) [1] is the de facto standard to build military Software Defined Radio (SDR) radios [2]. Several hundred thousand SCA military radios have been deployed worldwide by several nations. The SCA offers a component-based operating environment for heterogeneous embedded systems that ensures applications are portable across platforms made of different combinations of processors, operating systems, micro-kernels, and communication buses.

Embedded system applications, especially signal processing applications, are often optimized using target-specific instruction sets. Computing devices such as GPPs, DSPs, FPGAs, and GPUs provide various specialized instructions sets to accelerate data processing. While such an approach can provide great performance improvements, it decreases the level of portability of SCA components. Once a component has been optimized using a specific instruction set, it cannot be ported to another processor family without making changes to the source code.

OpenCL is an open and royalty-free standard maintained by a non-profit technology consortium called the Khronos Group [3]. Apple created the first version of OpenCL [4] to allow high-performance applications to be portable to various platforms. It is now supported by a very large number of commercial vendors [5]. OpenCL has been designed from the ground up for parallel processing. It supports task-based and data-based parallel programming which have been mapped to GPPs [6, 7, 8, 9], GPUs [10, 11, 12, 13], DSPs [14], and FPGAs [15, 16].

Creating SCA components using OpenCL to perform signal processing on various processors has been demonstrated to work [17]. Thanks to its support for a wide range of processors, OpenCL can significantly increase the level of portability of SCA applications. The current paper describes how the technique described in [17] can be improved. In particular, it examines the cost of moving data across SCA components. The paper proposes a novel approach that can provide better performances for SCA applications that use OpenCL.

2. USING OPENCL WITH SCA COMPONENTS

This section offers a brief overview of what SCA components are and how they are used to create applications. It follows with a description of OpenCL and how it can be used to create SCA components.

2.1. SCA Components and Applications

An SCA component is to software what an integrated circuit component (also known as a chip) is to hardware. They are components that can be reused from one system to another. Much like a hardware designer cannot modify a chip to change its behavior, a software developer does not modify the source code of a SCA component to change its behavior. The designer of the component must provide an interface that allows the user of a component to control it.

With hardware, chips offer pins that implement specific protocols to allow users to control them. With software, SCA components offer ports and properties as a means to interact with other software. Ports implement well-defined interfaces that can be used to exchange data and control the component. Properties provide a well-defined interface to allow a user to change the value of an attribute in order to alter the behavior of a component.

When a designer cannot find a chip that does exactly what is needed, he can connect several chips together to obtain the overall required behavior. Alternatively, the designer can create an Application Specific Integrated Circuit (ASIC). The same is true about SCA software components. The developer can assemble several software components together or create a new component.

SCA systems are made of two types of components: components that control the hardware and components that implement the application behavior. In the case of an SDR platform, some SCA components control hardware such as the transmitter, the transceiver, and the power amplifier. Other components perform the signal processing required to implement communication standards such as AM, FM, and LTE.

SCA applications are typically made of a series of interconnected software components that process digitized signals to, for instance, implement a communication standard, control a robot, or perform diagnostics. SCA application components interact with components that abstract the physical hardware. With a proper hardware abstraction layer, SCA applications become very portable across different physical platforms.

SCA components come as a set of files that contain binary code and meta-data. The SCA defines a standard set of meta-data that describes the different ports and properties of a component. A port is described in terms of the interface it provides or uses. Properties are described in terms of their data types and structures. SCA components are meant to support several different operating environments each. As such, the meta-data also describes which binary files are to be used to different operating environments. An operating environment represents an execution host and can be defined by a processor type, an operating system, a vector engine, or anything a component implementation might need to execute. SCA components can be executed on processors such as GPPs, DSPs, FPGAs, GPUs, etc. However, each application must have at least one component that runs on an operating system that supports a minimum set of POSIX calls.

2.2. OpenCL

Software developers use OpenCL to improve data processing performances by allowing task-based and data-based parallel programming. An OpenCL application is made of two parts: kernels and a host program. OpenCL kernels are routines (algorithms) that numerically process data.

Kernels are implemented in a C-like language with vector types and operations, synchronization, and functions that facilitate the use of parallelism by allowing the processing to be divided into work-items and work-groups. Kernels can be dispatched to run in parallel on a target device that supports OpenCL.

The host program runs on the host processor and is implemented using the OpenCL Application Programming Interface (API) to launch kernels on the target devices and manage device memory. The host processor is typically a GPP while the target device can be a GPP, a DSP, a GPU, or even an FPGA.

An OpenCL framework consists of a library that implements the host APIs, and an OpenCL compiler that compiles and links kernels for a specific target device. An OpenCL host program goes through the following steps for each buffer of data that needs to be processed. It uses the host APIs to first load the required OpenCL kernels onto the selected compute device. The host program then copies the data that needs to be processed from the host memory to the target device memory. Next, the host program launches the execution of the kernels and finally, it copies the processed data back from the target device memory to the host memory.

Conceptually, OpenCL offers nothing more than a co-processor. The novelty with OpenCL is that it supports a wide range of target devices as co-processors. OpenCL does so with a standard set of host and ‘C’ APIs which are supported by a very large segment of the high-tech industry manufacturers that includes Apple, Intel, AMD, ARM holdings, IBM, Qualcomm, Samsung, Creative Technologies, Altera, Xilinx, and more. Using the same

APIs for any target device provides a very high level of reusability.

2.2. A New Kind of SCA Component

Using OpenCL, the host program sets up the environment to exploit a co-processor to perform data processing. Using the SCA, at least one component needs to run on a host processor. The fundamental concepts behind SCA and OpenCL go hand-in-hand. It's easy to make an SCA component play the role of the OpenCL host program and dispatch the execution of signal processing functions to co-processors. The signal processing functions take the form of OpenCL kernels that get loaded onto target devices.

When an SCA component gets launched and initialized, it needs to look for and select an OpenCL target device. For each buffer of input data that the SCA component receives, it must load the kernels onto the selected target device, copy the data from the host to the target memory, launch the execution of kernels, and copy the data back from the target to the host memory. Once the processed data is back into the host memory, the SCA component must send the data to the next SCA component of the SCA application for further processing.

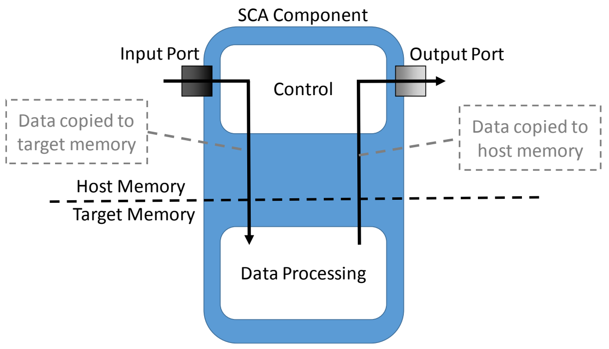

Figure 1 - Structure of an SCA Component that uses OpenCL

Figure 1 illustrates the structure of a typical SCA component. The main body of an SCA component is made of two main logical parts: the control part and data processing part. The black arrows represent the flow of data. Typically, the input data flows through a port and the control part routes the data processing part. The control part gets the data back from the processing part and sends the processed data to another component through an output port. In the case of an SCA component that uses OpenCL, the data processing part is actually made of kernels that run on the target processor.

Using OpenCL as described in this paper, the level of portability of an SCA component can be significantly increased. Without OpenCL, SCA components are often implemented using processor-specific instruction sets to increase performance. However, doing so makes the components very target specific. Thanks to OpenCL, the source code that performs data processing remains very portable across different types of target processors.

3. SCA APPLICATIONS AND OPENCL

Creating an SCA application from components that use OpenCL offers performances with an unprecedented level of portability across heterogeneous platforms. Target devices are typically orders of magnitude faster than a GPP for data processing [18, 19]. But since there is a cost associated with copying the data, the developer must ensure the increase in speed of processing more than compensates for the time it takes to copy the data to the target device and back. In cases where the data processing is not intensive enough, especially with large amounts of data, using an OpenCL target device may in fact be slower than using the host processor [20].

In the context of SDR, the data processing is very intensive and can benefit from OpenCL. With applications made of several components that use OpenCL, it is possible for more than one component to share the same OpenCL target device. That is especially true when an SCA application runs on a handheld platform. Indeed, several portable platforms are made of a single System-on-Chip (SoC) processor which consists of a GPP associated with a GPU or a DSP. Several cell phones and military handheld radios are designed around SoC processors. Desktop computers are also made of a single host processor and often only have one GPU card.

Running several SCA components on the same SoC processor leads to a situation where the data being exchanged is copied back-and-forth between the host and target memory. The more components use the same OpenCL target device, the more copies are done. Figure 2 illustrates this situation.

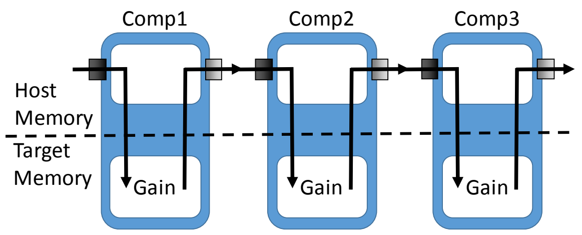

Figure 2 – Data Flow without any Enhancement

4. EXCHANGING DATA MORE EFFICIENTLY

Since copying the data between the host and target memory has a cost, it would make sense to avoid making a copy between each component of an application that uses the same target device. Figure 3 illustrates how the data could flow without being transferred to the host processor in between each component. From the figure, it is easy to see how performances could be increased especially with an application made of many more components that use the same OpenCL target device.

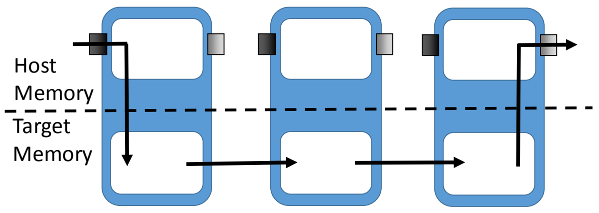

Figure 3 - Data Flow with Enhancement

One approach to avoid making unnecessary data copies is to create a new component that combines all the data processing algorithms into a single component. While this approach would minimize the number of copy operations, it violates the fundamental rules of component reuse. Creating new components for every situation requires more lines of code and reduces the potential for reusability. If the individual software components were actually hardware chips, a designer would almost certainly not combine them into a new chip. The cost of doing so could be high.

Another approach is to allow a component to determine when it is being connected to another component that uses the same OpenCL target device. When both components are colocated, they can use a more efficient way of exchanging data. Implementing this solution consists in conceiving a way to let two OpenCL kernels communicate directly with each other even when they are deployed by two separate SCA components. The next section identifies the requirements associated with communications between kernels.

4.1. The OpenCL Platform

The OpenCL platform is a concept that describes a system as being made of a number of compute devices connected to a host processor. The host processor is generally a GPP while the compute devices can be GPPs, GPUs, DSPs, and even FPGAs. The host processor is responsible for dispatching kernels to be executed on a compute device (i.e. the target device). In terms of OpenCL, a compute device is made of one or more compute units each of which contain many processing elements (PEs). Each kernel can be dispatched to run on all or many PEs in parallel.

4.2. The Host Program

The host program is responsible for scheduling the execution of kernels on various compute devices. It starts by getting a list of the available compute devices and chooses which devices it will target for the execution of kernels. In OpenCL terminology, the host program creates an OpenCL context from the list of the selected target devices. The context is then used to create a command queue that will be used to schedule kernels for execution.

To feed input data to a kernel, the host program needs to create an OpenCL buffer that is linked to the input data located in the host memory. Such buffers are created using the context where the kernels will be executed. The OpenCL framework automatically takes care of making the data available on the specific target device that will run the kernel.

For two kernels to share data efficiently, the same context must be used to schedule the execution of both kernels. This way, the host program does not need to get involved in copying data from one context to another after kernel executions. In fact, the most efficient way to share data is to run the kernels not only using the same context, but using the same target device. This way, even the OpenCL framework does need to get involved in copying the data from one target device to another.

Launching the execution of a kernel involves a few steps. For each kernel, the host program must read the kernel file to create an OpenCL construct that will be used for execution. The host program must also create all the OpenCL input and output buffer(s) the kernel will need. The host program schedules a kernel for execution by adding the kernel to a command queue and by specifying which input and output OpenCL buffer(s) the kernel will need. The OpenCL framework takes over at that point to make the buffers available on the target device that will run the kernel.

When several kernels are scheduled for execution, the most basic way to ensure an order of execution is to use an ordered command queue. This way, the kernels will be executed in the same order as the host program adds them to the command queue. Orderly execution is important to ensure output buffers are produced before they are used as input buffers by subsequent kernels. After the execution of the last scheduled kernel, the host program can copy the data from the OpenCL buffer back to the host memory.

Because command queues and buffers belong to a context, for kernels that interact with other kernels, it is more efficient to schedule the execution using the same context. An OpenCL context is an opaque data type that is allocated in a program space. As such, a context cannot be shared across multiple program spaces without using shared memory or similar technics. This is a very important characteristic that influences how an OpenCL context can be used in the context of an SCA system.

5. COLOCATION OF SCA COMPONENTS THAT USE OPENCL

The SCA is very flexible regarding how software components get executed. The version 2.2.2 of the SCA specification [21] supports the concept of running software components into individual process spaces. It can also run several components into a single process space. The latest SCA specification, version 4.1[23], supports even more options to colocate several SCA components.

Running several SCA components into a same process space allows each component to share a same OpenCL context. As explained above, sharing the same context allows kernels to interact without having the data go through a round trip via the host memory. Avoiding the un-necessary copies of the data across the bus between the host process and the target processor represents a substantial improvement over the technique proposed in [17].

The technique proposed in this paper allows components to determine when their kernels run on the same OpenCL context. The key element of the proposal exploits the connection process that creates a link between two components. With the SCA, a connection between two components is always unidirectional. SCA connections are nothing more than a mechanism to exchange references to components. This means that if Component1 is connected to Component2, Component1 can uses service of Component2 but not the other way around. To create a bi-directional relationship requires two connections, one in each direction.

A SCA connection provides a reference to an implementation of a specific interface described using CORBA's Interface Definition Language (IDL). The reference can therefore be used to invoke any operation defined by the IDL. The technique described in this paper relies on an IDL interface that allows a component to learn about the operating environment of another component to which it is connected. The detailed description of the interface is irrelevant as it can be implemented in multiple ways. The important aspect is the services the IDL interface must provide. Figure 4 illustrates an informal UML sequence diagram that describes the steps involved to allow a component to determine if another component uses the same OpenCL context or not. It also illustrates that the component that feeds another component does not copy data back to the host but instead provides the OpenCL buffer (transition #7).

The whole sequence of operations starts with Comp1 being connected with Comp2. The first thing that Comp1 does after being connected is verify if Comp2 implements the special OpenCL IDL interface (transition #2). If that is the case, Comp1 uses that interface to verify if it runs on the same processor as Comp2 and if they use the same OpenCL context (transition #3). This can be accomplished by comparing some specific identifiers.

Once Comp1 determines it shares the same OpenCL context with Comp2, it schedules its kernels for execution (transition #5) and waits for them to produce output buffers (transition #6). As soon as the OpenCL output buffers have been produced, Comp1 uses the OpenCL IDL interface to provide Comp2 with a reference to the new output buffers it has produced (transition #7). At that point, Comp2 schedules its kernels using the Comp1 output buffers as input buffers (transition #8). And this process repeats for each new output buffer Comp1 produces. This approach can be used between any two SCA components using the special OpenCL IDL interface.

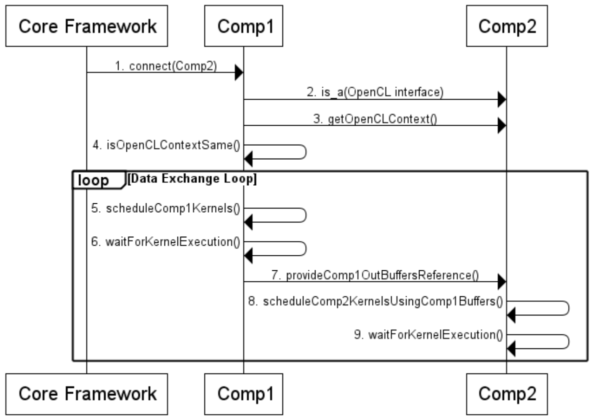
Core Framework
Comp1
Comp2
1. connect(Comp2)
2. is_a(OpenCL interface)
3. getOpenCLContext()
4. isOpenCLContextSame()
loop
[Data Exchange Loop]
5. scheduleComp1Kernels()
6. waitForKernelExecution()
7. provideComp1OutBuffersReference()
8. scheduleComp2KernelsUsingComp1Buffers()
9. waitForKernelExecution()
Core Framework
Comp1
Comp2

Figure 4 - Sequence of Interactions to Detect Colocation

When a component involved in a connection does not support OpenCL (or is running on a remote processor), the output buffers need to be copied back from the target memory to the host memory before they get delivered via CORBA (or another specialized protocol). This use case is the one described in [17].

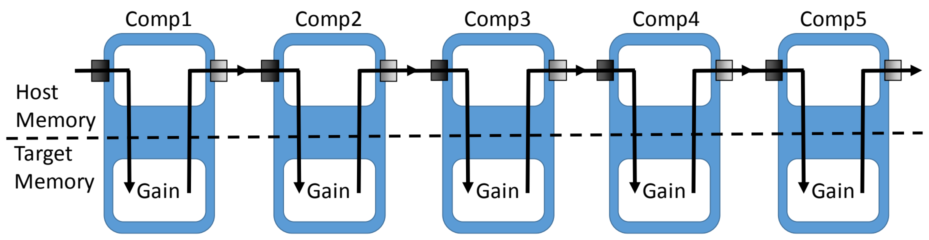

Figure 5 - Baseline Test Case: data goes back to host memory

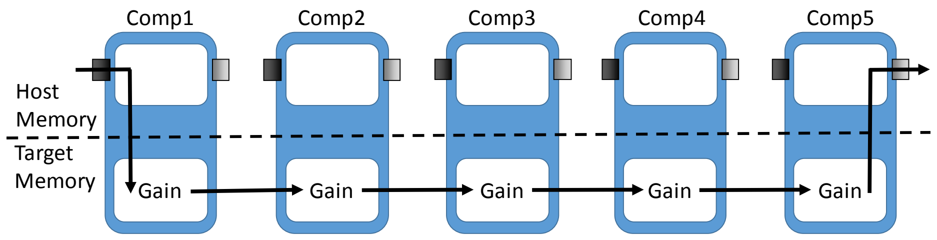

Figure 6 - Enhanced Test Case: data remains in target memory

6. PERFORMANCE METRICS

To perform our experiments, 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 one OpenCL compute device. The device is a PCI-Express 3.0 NVIDIA GeForce GT 635 GPU card. We used the NVIDIA OpenCL CUDA 7.0.41 platform driver which supports OpenCL1.1.

Figure 5 and Figure 6 show the two test cases used to conduct all the experiments. It is important to note that both test cases rely on the same SCA application. The application is made of five identical components which add gain by multiplying the input data by a set value. The experiments consist in measuring how long it takes to push data through all five components. The gain function takes a small and constant amount of time. Thus the variations in the measurements between the different experiments are caused by differences in the time it takes for the data to flow through the application.

In all the experiments, the application was launched such that all the SCA components are executed on the same processor. The difference is that in first test case (herein "baseline test case"), the components are launched as individual process spaces while in the second test case (herein "enhanced test case"), all the components where launched as part of the same process space. Thanks to the special OpenCL IDL interface, in all the experiments conducted for the enhanced test case, the components are able to detect they are running in a single process space. They can also determine that they are using the same OpenCL context. Consequently, the components do not copy output buffers back to the host computer but instead keep the data in the OpenCL target memory.

In all the experiments, the data buffers originate from a generator component (not shown in diagrams) running on the same processor. The generator component always runs in a process space of its own and uses a standard CORBA call to deliver the buffers to the first gain component (i.e. Comp1) of the application. The last gain component of the application (i.e. Comp5) copies the output buffers from the target memory back to the host memory and sends it to a sink component (not shown in diagram). In all the experiments, the sink component runs in a process space of its own. Data buffers are always delivered to the sink component via a standard CORBA call.

As data buffers flow through the application, each gain component inserts a time-stamp that records time of reception, right before any data processing is done. The components also insert a time-stamp after the output buffers are produced, right before the buffers are sent to the next component. Each component preserves the time stamps inserted by other components. The sink component accumulates all the time stamps and produces a report after all the buffers of the experiment have been processed.

Each test case was used to conduct experiments using 7 different buffer sizes: 2KB, 4KB, 32KB, 64KB, 256KB, 1024KB, and 4096KB. The buffers were made of 32 bits floating point values. Each experiment was conducted by sending one thousand buffers, one at the time. Said differently, the application never processed more than one buffer simultaneously. This was done in order to avoid side effects like concurrency, scheduling, and caching. Such side effects also have an impact on how long it takes to move data between the GPP and the GPU. However, the enhancement described in this paper can be used to process several buffers at the same time. Nonetheless, the goal of the paper is not to measure the raw data processing performances. Such measurements are highly dependent of the data processing algorithms that are being used in the application. The goal of the paper is to measure the amount of time that can be saved by avoiding the repeated copy of buffers between the host and target memory.

KB	Samples per Buffer	Baseline (µs)	Enhanced (µs)	Baseline / Enhanced
2	**512**	732	435	1.68
4	**1024**	696	442	1.57
32	**8192**	851	350	2.43
64	**16,384**	992	444	2.23
256	**65,536**	2527	529	4.8
1024	**262,144**	10,735	3039	3.5
4096	**1,048,576**	32,130	7076	4.5

Table 1 - Measurements for Both Test Cases

As explained in [22], when all the components of an application execute in a same process space, high performance Object Request Brokers (ORBs), like *ORBexpress RT* from Objective Interface Systems, optimize interactions by making native function calls as opposed to use a communications transport like TCP/IP. As a result, all the one-way interactions that are normally conducted on a separate thread are executed using the calling thread. This causes a thread of the first component to be used for the processing of all the components in the application. In order to avoid this issue, the SCA components for this paper use a queue to store incoming buffers and a separate thread to perform the processing.

In an effort to minimize the impact of threads being moved to a different core during the test, all the application components were assigned to one single core of the processor. Without using core assignment, the measurements are impacted by low-level cache misses which occur non-deterministically.

Table 1 contains two columns, one for each test case. The measurements represent the time it takes, in microseconds, for a data buffer to transition from the input of the first component (i.e. Comp1) to the output of the last component (i.e. Comp5) of the application. The "Baseline" column contains measurements that involves copying data buffers across the GPP/GPU bus in between each of the 5 components. This means each buffer is copied 10 times to go through the whole application. The "Enhanced" column contains measurements for the enhanced version of the application which involves only two copies of the buffers across the GPP/GPU bus. As the measurements show, the enhanced test case provides better performances. The measurements indicate the enhanced test case is 1.57 to 4.8 times faster because it spends less time copying data. Each measurement in the table represents the average, in microseconds, after running one thousand experiments.

7. CONCLUSION

This paper starts by describing that SCA components can be made more portable using OpenCL to implement the data processing part of a component. It briefly describes that OpenCL requires data to be copied from the host memory to the OpenCL target memory prior to perform the data processing. It also describes that SCA components must copy the data back to the host memory after processing in order to be able to route the data to the next SCA component.

This paper introduces a new technique to allow SCA components to avoid making copies across the memory bus when not required. The technique allows components that run from the same host processor and use the same OpenCL context to share data buffers directly. The technique is implemented in such a way that components remain very flexible. Components can autonomously determine when copies must be made and when they can be avoided.

Measurements provided clearly show that a significant amount of time can be saved by not making data do roundtrips over the bus connecting the host processor to the OpenCL target processor. The new technique described in this paper allows for up to 4.5 times faster data transfers between components which translates into better signal processing performances for a complete application.

Future work will involve investigating how the technique described in this paper performs when several buffers are being processed in parallel by different components of an SCA application. As demonstrated by the work performed for this paper, the notion of locality for two components can be exploited to increase performances significantly. The authors of this paper will engage in a process to generalize and propose the concept of locality detection for adoption in a future version of the SCA specification.

6. REFERENCES

[1] Joint Tactical Networking Center (JTNC), Software Communications Architecture, http://www.public.navy.mil/jtnc/sca/Pages/default.aspx

[2] SCA Standards for Defense Communications, Wireless Innovation Forum, http://www.wirelessinnovation.org/assets/sca%20standards%20%20global%20adoption%20version%201.0%20high%20res%20final.pdf

[3] Khronos Group, OpenCL Standard, https://www.khronos.org/opencl/

[4] OpenCL, Wikipedia, https://en.wikipedia.org/wiki/OpenCL

[5] Complete list of companies and OpenCL conformant products, The Khronos Group, https://www.khronos.org/conformance/adopters/conformant-products#opencl

[6] Getting Started with Intel® SDK for OpenCL™ Applications, Intel, https://software.intel.com/en-us/articles/getting-started-with-opencl-code-builder

[7] Getting Started with OpenCL™, AMD, http://developer.amd.com/tools-and-sdks/opencl-zone/opencl-resources/getting-started-with-opencl/

[8] ARM extends OpenCL to the ARM Cortex-A processor family, ARM, https://www.arm.com/about/newsroom/media-alert-arm-extends-opencl-to-the-arm-cortex-a-processor-family.php

[9] T. Ballard, “OpenCL for Linux on Power”, https://www.ibm.com/developerworks/community/wikis/home?lang=en#!/wiki/Wbf059a58a9b9_459d_aca4_493655c96370/page/OpenCL%20for%20Linux%20on%20Power

[10] OpenCL Platform with Intel® Graphics, Intel, https://software.intel.com/en-us/node/540387

[11] NVIDIA Adds OpenCL To Its Industry Leading GPU Computing Toolkit, Nvidia, http://www.nvidia.com/object/io_1228825271885.html

[12] L. Howes, OpenCL Parallel Computing for GPUs, AMD, https://developer.amd.com/wordpress/media/2012/10/OpenCL_Parallel_Computing_for_CPUs_and_GPUs_201003.pdf

[13] ARM Mali-T600 Series GPU OpenCL Developer Guide, ARM,http://infocenter.arm.com/help/topic/com.arm.doc.dui0538f/DUI0538F_mali_t600_opencl_dg.pdf

[14] Introduction - TI OpenCL Documentation, TI, http://downloads.ti.com/mctools/esd/docs/opencl/intro.html

[15] M. Parker, M. Jarvis, The Most Under-Rated Design Tool Ever, Altera, http://www.eetimes.com/author.asp?section_id=36&doc_id=1327664

[16] S. Leibso, OpenCL code compiled with Xilinx SDAccel accelerates genome sequencing, beats CPU/GPU performance/W by 12-21x, Xilinx, https://forums.xilinx.com/t5/Xcell-Daily-Blog/OpenCL-code-compiled-with-Xilinx-SDAccel-accelerates-genome/ba-p/680764

[17] S. Bernier, F. Levesque, M. Phisel, and D. Hagood, Using OpenCL to Increase SCA Application Portability, Proceedings of SDR-WInnComm-Europe 2015, September 2015.

[18] M. Papadimitriou, J. Cramwinckel, A.L. Varbanescu, Accelerating Computational Finance Simulations with OpenCL. Euro-Par 2016, 22nd International European Conference on Parallel and Distributed Computing.

[19] K. Li, M. Wu, G. Wang, J.R. Cavallaro, A High Performance GPU-based Software-defined Basestation, 48th IEEE Asilomar Conference on Signals, Systems, and Computers (ASILOMAR), 2014.

[20] A. Kalia, D. Zhou, M. Kaminsky, D. G. Andersen, Raising the Bar for Using GPUs in Software Packet Processing. Carnegie Mellon University and Intel Labs. USENIX NSDI 2015.

[21] SCA Specification version 2.2.2, Joint Tactical Networking Center, http://www.public.navy.mil/jtnc/SCA/Pages/sca1.aspx

[22] S. Bernier, H. Latour, J.P. Zamora, “How different messaging semantics can affect SCA applications performances: a benchmark comparison”, Analog Integrated Circuits and Signal Processing, Springer Verlag, December 2011.

[23] SCA Specification version 4.1, Joint Tactical Networking Center, http://www.public.navy.mil/jtnc/SCA/Pages/sca1.aspx