OpenCL: A suitable solution to simplify and unify High Performance Computing developments

Jonathan Passerat-Palmbach, David Hill

UMR 6158 CNRS

OpenCL: A suitable solution to simplify and unify High Performance Computing developments

Jonathan PASSERAT-PALMBACH[*†‡§],

David R.C. HILL [†‡§]

Originally published in: Patterns for Parallel Programming on GPUs *(chapter 8)*
— 2013 — pp 189–209
ISBN: 978-1-874672-57-9
Saxe-Coburg Publications

Abstract: Manycore architectures are now available in a wide range of HPC systems. Going from CPUs to GPUs and FPGAs, modern hardware accelerators can be exploited using heterogeneous software technologies. In this chapter, we study the inputs that OpenCL offers to High Performance Computing applications, as a solution to unify developments. In order to overcome the lack of native OpenCL support for some architectures, we survey the third-party research works that propose a source-to-source approach to transform OpenCL into other parallel programming languages. We use FPGAs as a case study, because of their dramatic OpenCL support compared to GPUs for instance. These transformation approaches could also lead to potential works in the Model Driven Engineering (MDE) field that we conceptualize on this work. Moreover, OpenCL's standard API is quite rough, thus we also introduce several APIs from the simple high-level binder to the source code generator that intend to ease and boost the development process of any OpenCL application.

Keywords: OpenCL; Abstraction layers; Survey; Source-to-source transformation; Manycore

[*] This author's PhD is partially funded by the Auvergne Regional Council and the FEDER
[†] ISIMA, Institut Supérieur d'Informatique, de Modélisation et de ses Appplications, BP 10125, F-63173 AUBIERE
[‡] Clermont Université, Université Blaise Pascal, LIMOS, BP 10448, F-63000 CLERMONT-FERRAND
[§] CNRS, UMR 6158, LIMOS, F-63173 AUBIERE

RESEARCH CENTRE
LIMOS - UMR CNRS 6158
Campus des Cézeaux
Bâtiment ISIMA
BP 10125 - 63173 Aubière Cedex
France

1 Introduction

At the end of the last decade we entered more deeply into the era of hybrid computing, where different types of processors are used in a common computing system. Modern accelerators are now available as COTS (Commercial Off The Shelf), we can buy and or design our own high performance hybrid architecture. Manycore CPUs (Central Processing Units), GPGPUs (General-Purpose Graphical Computing Units) and more recently the AMD APUs (Accelerated Processing Units) are gaining more and more interest. FPGAs have also been considered to be elements of a general purpose High Performance Computing system, they are used in some of the best architectures (like the one proposed by Cray Inc.) [5]. Nowadays, the High Performance Computing (HPC) community tends to champion GPU architectures to speed-up applications, but according to [16], FPGAs offer better performances than GPUs when faced with some integer arithmetic operations. In addition to impressive performances, the FPGA approaches have many advantages including small footprint with very low energy costs. Most of FPGA logic is configurable, and this architecture is saving the processing power since we only design the specific features that will be used. If we can reuse the same silicon for different functions, we also have the advantage of an easy mapping of different operations to different silicon thus allowing massive parallelism. Long before the GPGPU boom, in 2004, companies joined to establish the FPGA High Performance Computing Alliance (FHPCA). This alliance is dedicated to the use of Xilink FPGAs for high end real-world industrial applications. Since the advent of the Virtex 4 (and now with much larger devices) the use of FPGAs for general purpose numerical computing has become a reality. Even if it was not really noticed in the Top500 at that time, the University of Edinburgh supercomputing centre proposed in 2007 an FPGA based supercomputer named "Maxwell".

The case of FPGAs in HPC is particularly significant. Given the advantages previously exposed, the question is why don't we have more FPGAs in HPC? The main cause in our opinion is that the programming model problem is neither mature nor accessible to the scientific community. Common HPC developers are not used to programming in Hardware Description Languages (HDLs, such as VHDL) for FPGAs, and the problem is not just learning a new language, since it is mainly a design issue. Indeed, we do not design circuits as we design programs. Even if predefined libraries are available, they do not compensate the fact that there is no real standard hardware. The point is that we need a technology that can be understood by the heterogeneous HPC community, allowing them to address the wide range of hardware accelerators at their disposal.

Since its introduction in June 2008, the OpenCL™ (Open Computing Language) standard has been specifically designed to address the challenge of executing HPC applications on heterogeneous systems consisting of multi-core processors, GPGPUs and FPGAs. OpenCL is based on the C programming language but it provides additional features to leverage highly-parallel hardware accelerators. OpenCL allows the programmer to explicitly specify and control parallelism.

Though numerous accelerator vendors started to support OpenCL for CPUs and GPUs, fewer efforts were given to support hybrid systems with FPGAs. Recent advances from the Altera Corporation, make us confident that the OpenCL programming model will propose advantageous solutions to common problems that refrained the adoption of FPGAs for HPC. While native tools are not reliable yet, we can still focus on third-party developments that act as converters from OpenCL to HDL without the need of any further development from FPGAs vendors.

Part of this work actually considers FPGAs as a case study of the ability to take advantage of OpenCL on platforms that do not officially support this technology. We have chosen to tackle this problem through FPGA-aimed proposals, since this community has been very productive in terms of solutions to detach from vendors. Still, the same conclusions could be applied to other architectures lacking OpenCL support, the most notable being Intel Sandy/Ivybridge GPU parts, when used under the Linux OS (Operating System).

Vendor-agnostic approaches should democratize the use of OpenCL for HPC developments, and

promote GPU computing at the same time, as the main platform offering OpenCL support. However, OpenCL suffers from a highly verbose and constrained API. Thus, this study also surveys software solutions assisting OpenCL development.

In this chapter, we:

- State the need of an abstraction layer for OpenCL;
- Survey the propositions to provide OpenCL support for any architecture through a source-to-source transformations approach;
- Study the most significant Application Programming Interfaces (APIs) to ease OpenCL development and compare their features;
- Recall the inputs of the OpenCL technology in High Performance Computing applications, and especially for GPUs.

2 On the need for an abstraction layer for OpenCL

2.1 OpenCL in brief

OpenCL is a standard proposed by the Khronos group that aims to unify developments on various kinds of hardware accelerators architectures like CPUs, GPUs and FPGAs. It provides programming constructs based upon C99 to write the actual parallel code (called the kernel). Kernels execution is executed by several work-items, that will be mapped to different execution units depending on the target: for instance, GPUs will associate them to local threads. For scheduling purposes, work-items are then bundled into work-groups each containing an equivalent amount of work-items.

They are enhanced by APIs (Application Programming Interface) used to control the device and the execution. At the time of writing, the latest version of the API is 1.2 [7] that has been released in November, 2011. OpenCL programs execution relies on specific drivers issued by the manufacturer of the hardware they run on. The point is OpenCL kernels are not compiled with the rest of the application, but on the fly at runtime. This allows specific tuning of the binary for the current platform.

OpenCL brings three major inputs to HPC developments: as a cross-platform standard, it allows developing applications once and for all for any supported architecture. It is also a great abstraction layer that lets developers concentrate on the parallelization of their algorithm, and leave the device specific mechanics to the driver.

2.2 OpenCL: a constrained API

In the introduction we have evoked the major problem of OpenCL: its complicated API. Concretely this lies in three major problems that will be detailed hereafter: bloated source code, double calls and risky constructs. First, when kernel functions that execute on the hardware accelerator remain concise and thus fully expressive, host API routines result in verbose source code where it is difficult for an external reader or for a non-regular developer, to determine the purpose of the application among all those lines. Second, some design choices of the API make it even more verbose when trying to write portable code that can run on several hosts without any change. For instance, OpenCL developers are used to calling most of the API query functions twice. Indeed, a first call is often required to determine the number of results to be expected, and a second actually gets the right amount of elements. We will designate these two subsequent invocations of the same routine as the **double-call** pattern hereafter. Finally, such verbose constructs discourage developers to check the results of each of their calls to the API. At the same time, the OpenCL API is very permissive with regards to the type of the parameters its functions

accept. Now imagine that two parameters have been awkwardly swapped in an API call, it is very likely that the application keeps quiet about this error if the developer has not explicitly checked the error code returned by this call. In an heterogeneous environment such as OpenCL, where the parallel part of the computation will be relocated on hardware accelerators, runtime errors that only issues error codes are not the easiest bugs to get rid of.

All these drawbacks make it clear that although the OpenCL standard is a great tool offering portability in high performance computing, it does not meet the higher level APIs expected by parallel developers to help them avoid common mistakes, and to produce efficient applications in a reasonable lapse of time.

3 OpenCL support on various platforms: the FPGAs case study

Enabling OpenCL for a particular FPGA architecture can be achieved in two major ways. The first relies on vendors who need to provide an OpenCL driver implementation for their particular device. When both NVIDIA and ATi provide OpenCL drivers for their GPU products, Intel has recently enabled a large set of its processors to support OpenCL. However the driver implementation solution is still in its infancy across the FPGA community. Altera is known for their FPGA products, but is also an active member of the Khronos Group, responsible of the OpenCL standard. The company has announced the development of an OpenCL framework, and presents encouraging results of a Monte-Carlo Black Scholes simulation implementation in [1].

Researchers' initiatives have led us to consider another way of thinking that does not involve any intervention from the FPGA's vendor, but relies on third-party developments to provide OpenCL support for FPGAs. The main advantage of these solutions is that they can be applied to any platform lacking OpenCL support. In this section, we study the most promising solutions from the literature, focusing on FPGAs, as long as this community has been in need of a unified programming model for a long time now, and proposed consequently more tools to meet their expectations.

3.1 Source-to-source transformation approach

Supporting the whole OpenCL standard can become a long and costly operation for a vendor, while its community might not perceive the benefits of switching to OpenCL developments, or simply do not require high performance. That being said, another strategy comes into play to bypass vendors decisions: offer OpenCL support for a range of devices from a third-party effort. Such projects are all based on the same principle that consists in transforming the OpenCL source code into a language already supported by the target platform. In this section, we will survey the most interesting proposals of source-to-source compiling aiming at similar goals.

3.1.1 At the beginning, there was CUDA

At the moment, OpenCL is mostly known to be a unified solution to program GPUs from various origins, and to compete with NVIDIA's CUDA technology in the GPU computing domain. Being almost two years older than OpenCL, CUDA has been targeted by research works earlier than its counterpart, particularly when considering the opportunities to have it running on other devices than NVIDIA GPUs. Most of these approaches make use of the same principle as the strategy we identified to provide OpenCL support for an FPGA architecture from a third-party initiative: i.e. source-to-source transformation.

In 2008, Stratton et al. proposed MCUDA, a tool intended to execute CUDA code on a multi-core CPU [15]. This paper brings up the fundamental problems that will be encountered when attempting to set up such a code generation approach, and even puts forward efficient solutions. Let us recall that CUDA relies on the Single Instruction, Multiple Thread (SIMT) paradigm, so the application logic is placed at the thread level. Threads perform the same operations on different input data, but sometimes,

an operation needs to access data that have been processed by another thread. In such a case, CUDA developers have to synchronize their threads at the particular step of the algorithm dealing with shared data to make sure the data they are processing is up to date.

Before parallelizing the application among CPU threads, MCUDA first serializes the CUDA input in order to analyse it more simply. MCUDA tries to identify independent code parts in terms of data within the newly serialized code. When such parts are detected, they are bounded by what is called a thread-loop. Thread-loops induce implicit synchronization points at their boundaries. Thus, an explicit request to synchronize threads in the original CUDA source code will result in two thread-loops: one embracing the code before the synchronization point and the other one after.

Instead of basing the computation on logical threads, MCUDA rises to a block of logical threads level. This strategy excludes semantic losses since blocks are independent along a classical CUDA execution. They contain logical threads that will perform the same operations with no constraint on the order in which the blocks are processed. MCUDA behaves identically, since after being serialized, blocks are executed by OS threads created by OpenMP directives, and thus relying on the underlying scheduling policy for their order of execution to be determined.

Now that the code is serialized and that the synchronization points are respected, MCUDA treats memory accesses. The most important point of this step is to understand that CUDA, such as OpenCL, defines several data areas mainly to take advantage of the different caching mechanisms available in a GPU architecture. As long as CPUs display a single cached memory space, MCUDA maps any kind of memory to the unique memory space of the CPU. Still, memory areas also bring information on the scope of the data they store, forcing MCUDA to apply another conversion step that again prevents semantic losses in the output code. For example, when variables shared by all the logical threads of a block can be replaced by a single variable, elements declared as private to a thread have to be duplicated. To do so, MCUDA introduces arrays containing the particular value of a given thread in the cell corresponding to its identifier. We will see further on that the same pondering about memory spaces must be observed when thinking about an FPGA implementation.

Basically, the philosophy of MCUDA lies in two main steps:

1. Group local-thread based computations and express them at a block level (SIMT $\rightarrow$ SIMB);

2. Parallelize the execution of independent blocks with a pool of worker threads.

More generally, the purpose is to first obtain independent elements before executing them concurrently following an arbitrary scheduling. This mechanism is classical and can also be found in other works involving CUDA and GPU architectures such as [13]. This implies two major choices when implementing an OpenCL converter: first, you need to figure out how the work-items will be mapped on the target architecture, and second, how the memory hierarchy will be represented on your FPGA.

One year later, some of the authors of MCUDA applied their conclusions to FPGA architectures and proposed FCUDA: a tool to compile CUDA kernels onto FPGAs. FCUDA's philosophy is quite similar to MCUDA's in the fact that its main purpose is to translate CUDA SIMT code into SIMB independent blocks that are ready to be transformed. Now, when MCUDA spreads those blocks on multiple CPU cores through OpenMP threads, FCUDA will synthesise FPGA cores according to the Register Transfer Level (RTL) design corresponding to the kernel function.

The way FCUDA deals with memory is quite interesting and thoroughly described in the paper [12], wherein authors make concrete proposals concerning the memory locations of the FPGA assigned to their CUDA counterparts. For instance, they propose to take advantage of Direct Memory Access (DMA) burst mode in order to implement constant memory.

Additionally, the workflow allowing FCUDA to transform CUDA code into FPGA Hardware Description Language (HDL) requires an intermediate stage where C code is produced and passed to a third-party software. This step enables developers to tune the resulting code before it is synthesized into

a RTL design. However, we need to note that FCUDA displays some drawbacks, indeed it only handles kernels but does not take into account the host API.

3.1.2 Then comes OpenCL

The MCUDA way to deal with parallel applications was pursued by several recent studies, this time generating HDL for a given FPGA from OpenCL source code. In 2011, Owaida et al. introduce SOpenCL (Silicon OpenCL) a source-to-source transformation process that converts OpenCL code into HDL to build FPGA-enabled hardware accelerators [11]. Just like MCUDA, SOpenCL first applies a serialization step on the kernel in order to join logical threads from a given work-group into independent execution environments. SOpenCL uses the fact that work-groups are actually 3D containers of local-threads to insert a triple-nested loop that will pass on each local-thread thanks to its *x, y, z* coordinates in the kernel to compute the data it was assigned.

Barriers and synchronization points are also handled in a similar way. Whenever a synchronization point must be set up, the loop around the current statement is split into two sub-loops: the first deals with the part of the statement before the synchronization point, while the second handles the remaining part. This process is called **loop fission**.

Finally, memory areas are also treated in a way close to MCUDA. The idea is to figure out the lifetime of a variable to state whether it can be reused across loops or not. Concerning the variables' scope, each logical-thread is provided with a private copy of the variable only if the latter is meant to be used in several sub-loops, which would potentially overwrite its content. The final location of variables in memory is bound to the targeted FPGA, but we have seen previously that no semantic information was lost during this stage.

What makes SOpenCL original is its template-based approach when the time comes to generate HDL code. SOpenCL proposes a hardware accelerator template including both the Data Path and Stream Units that will be involved in the computation. Taking advantage of the LLVM compiler framework [9], the hardware generation flow results in synthesisable HDL for the accelerator following an architectural template, instead of issuing an unpredictable output code.

In 2010, Jääskeläinen et al. proposed another work inspired from MCUDA. However, this OpenCL-based design methodology [6] differs from other proposals by exploring the extension feature offered by the OpenCL standard. As we have seen previously, OpenCL allows vendors to add specific extensions from their own to exploit their hardware a bit further than the sole standard API would. The main advantage of this capacity is that it does not imply rewriting code when switching from an extension-enabled architecture to a standard one. Thus, vendors can allow to leverage the specificities of their hardware, while remaining compliant with others'. As long as extensions availability can be checked at source level, one can adapt its algorithm behaviour to take advantage of the extensions only when possible. In an FPGA approach, this feature appears crucial since it lets hardware designers provide custom architectures synthesized on an FPGA platform. Clients can then choose to benefit from these particular developments or not, and even try both solutions in the blink of an eye without having to revise their entire source.

In conclusion, we have seen that no matter the source-to-source tool we studied, the implementation choices are roughly the same. The process follows a common skeleton that is summed up hereafter:

1. Serialize work-groups by executing their local-threads sequentially;
2. Eliminate barriers and synchronization points thanks to loop fission;
3. Analyse variables livelihood to determine which can be shared and which should be copied in a private area for each local-thread;

4. Convert the resulting source code to an HDL, taking work-groups as the base unit for the computation.

4 High-Level APIs for OpenCL: Two Philosophies

4.1 Ease OpenCL development through high-level APIs

4.1.1 Standard API C++ Wrapper

The OpenCL standard not only describes the C API that any implementation should meet, but also the C++ wrapper that comes along with it. This wrapper provides both C++ bindings of the OpenCL calls, and also two restricted declinations of the Standard Template Library (STL) from genuine C++: the string and vector classes. These two classes have been rewritten for the purpose of the OpenCL wrapper as a subset of their counterparts but are mostly compliant. The idea is to get rid of STL's bloated classes, which *std::string* is nothing but the best example. As long as they display an interface close to the original classes, the OpenCL versions, stored in the *cl* namespace, can be swapped with the matching STL class thanks to a single macro.

Another C++ mechanism nicely leveraged by the OpenCL C++ wrapper is exceptions. OpenCL errors are handled in a low-level way with error codes to be compared to 0 to assess whether the previous call is completed successfully. Obviously this technique is not quite adapted to develop with higher level languages, because it inflates source codes with non-effective lines. Moreover, it forces developers to be very careful and to explicitly check the returning code of their invocations, and, in case of problems, to retrieve the corresponding error. In our case, this painful process is handled by the exception mechanism that forces developers to catch the exception and treat it consequently.

The remaining elements of this binding are dedicated to ease the OpenCL API for developers. The most significant example in this way is the double call pattern foreseen in our short introduction to OpenCL in section 2.2. Using the traditional OpenCL C API, one needs to perform two successive calls to the same function to first figure out the number of results to be stored in an array provided for this purpose, before actually filling it through a second call requesting the exact number of elements to store in the array of results subsequently. The C++ wrapper here highly facilitates the process since a single call is needed to obtain the results, while *cl::vector* is used instead of dynamically allocated arrays.

Listing 1 shows an example of the wrapper syntax. Although efforts can be noticed to provide a simple API, the result remains quite verbose since Listing 1 only lists GPU devices and creates a dummy kernel from a source file in the context of the discovered device:

Listing 1: GPU devices listing and kernel creation using the C++ wrapper API (adapted from [14])

```
try {
    // Place the GPU devices of the first platform into a context
    cl::vector<cl::Platform> platforms;
    cl::vector<cl::Device> devices;

    cl::Platform::get(&platforms);
    platforms[0].getDevices(CL_DEVICE_TYPE_GPU, &devices);
    cl::Context context(devices);

    // Create and build program
    std::ifstream programFile("kernel.cl");
    std::string programString(std::istreambuf_iterator<char>(programFile),
```

```
            (std::istreambuf_iterator<char>()));
        cl::Program::Sources source(1, std::make_pair(programString.c_str(),
            programString.length()+1));
        cl::Program program(context, source);
        program.build(devices);

        // Add kernel to progam
        cl::Kernel fooKernel(program, "foo");

        // Create kernel
        cl::vector<cl::Kernel> allKernels;
        program.createKernels(&allKernels);
          }
    catch(cl::Error e) {
        std::cerr << e.what() << std::endl;
    }
```

4.1.2 QtOpenCL

Used to provide nice bindings for C++ development tools going from database drivers to concurrency, the Qt library developed by Nokia [10] now also offers its own OpenCL wrapper as a set of extensions named QtOpenCL. At the time of writing, we would like to insist on the fact that QtOpenCL is neither included in the stable Qt 4.7 release, nor in the future 4.8 version according to Nokia's roadmap. However, QtOpenCL is freely available for download as an extension and is compatible with Qt 4.6.2 and Qt 4.7 frameworks.

As an OpenCL wrapper, QtOpenCL aims to break down three main barriers that we already identified as OpenCL drawbacks: initialization phase, memory management and kernel execution. To assist users in the initialization phase, and particularly to compile the external OpenCL source code, QtOpenCL handles automatically OpenCL source files reading through a single method. Memory management receives as much consideration so that buffers can be created more simply. Moreover, their object oriented interface allow users to call methods such as *read()* directly on buffers to retrieve results from a hardware accelerator.

The third point of the QtOpenCL intentions targets kernel executions. This aspect is handled "à la Qt" thanks to the *QFuture* class, well-known by developers using Qt for their CPU developments involving concurrency.

QFuture is an event returned when a kernel is run through the *QtConcurrent::run* method, and allows to wait for the kernel to complete. In addition, *QFuture* is compatible with the *QFutureWatcher* class that uses the signal/slot mechanism which Qt is based upon. The latter mechanism is an implementation of the Observer design pattern that causes a routine to be called when a particular event occurs. In our case, the event signals the completion of the associated kernel and can rise an update of the Graphical User Interface (GUI) of the application for example.

Listing 2 presents how to enqueue a dummy kernel that takes a vector of integers as a parameter and outputs another one, using QtOpenCL:

Listing 2: GPU context creation kernel enqueueing using QtOpenCL

```
// context creation
QCLContext context;
if ( !context.create() ) {
        std::cerr << "Error in context creation for the GPU" << std::endl;
```

```
        return 1;
}

const int vectorSize = 1024000;
QCLVector<int> inVector = context.createVector<int> ( vectorSize );
QCLVector<int> outVector = context.createVector<int> ( vectorSize );

for (int i = 0; i < vectorSize; ++i) {
        inVector[i] = i;
}

// kernel build
QCLProgram program = context.buildProgramFromSourceFile ( "./kernel.cl" );
QCLKernel kernel = program.createKernel ( "foo" );

// enqueue and run kernel
kernel.setGlobalWorkSize ( vectorSize );
kernel ( inVector, outVector );
```

4.1.3 PyOpenCL

PyOpenCL is a research initiative developed at the same time as its CUDA counterpart PyCUDA [8]. Both approaches rely on a concept called GPU Run Time Code Generation (RTCG) that intends to solve common problems encountered when harnessing hardware accelerators. In PyOpenCL, this translates in a flexible code generation mechanism that can adapt to new requirements transparently. In the end, programmers can summon kernels as if they were methods from the program instance they have just built.

Apart from those dynamic features, PyOpenCL displays the classical inputs from an OpenCL wrapper developed in a high level language such as Python. The latter being well-known for its concision and simplicity, PyOpenCL directly benefits from this characteristic. This is achieved widely because of Python dynamic typing system that delegates an important part of the work to the interpreter. Finally, OpenCL source code reading takes advantage of Python file handling capacities, so that a program can be read from source and built using no more than four lines, as it can be seen in Listing 3:

Listing 3: GPU program building using PyOpenCL

```
programFile = open ( 'foo.cl', 'r' )
programText = programFile.read ( )
program = cl.Program ( context, programText )
program.build ( )
```

4.2 Generating OpenCL source code from high-level APIs

4.2.1 ScalaCL

ScalaCL is a project, part of the free and open-source Nativelibs4java initiative, led by Olivier Chafik [4]. The Nativelibs4java project is an ambitious bundle of libraries trying to allow users to take advantage of various native binaries in a Java environment.

From ScalaCL itself, two projects have recently emerged. The first one is named Scalaxy. It is a plugin for the Scala compiler that intends to optimize Scala code at compile time. Indeed, Scala functional

constructs might run slower than their classical Java equivalents. Scalaxy deals with this problem by preprocessing Scala code to replace some constructions by more efficient ones. Basically, this plugin intends to transform Scala loop-like calls such as map or foreach by their while loops equivalents. The main advantage of this tool is that it is applicable to any Scala code, without relying on any hardware.

The second element resulting from this fork is the ScalaCL collections. It consists in a set of collections that support a restricted amount of Scala functions. However, these functions can be mapped at compile time to their OpenCL equivalents. A compiler plugin dedicated to OpenCL generation is called at compile time to normalize the code of the closure applied to the collection. Functional programming usually defines a closure as an anonymous function embedded in the body of another function. A closure can also access the variables from the calling host function. The resulting source is then converted to an OpenCL kernel. At runtime, another part of ScalaCL comes into play, since the rest of the OpenCL code, like the initializations, is coupled to the previously generated kernel to form the whole parallel application. The body of the closure will be computed by an OpenCL Processing Element (PE), which can be a thread or a core depending on the host where the program is being run. Listing 4 presents a simple closure that computes the apply the cosine function to every element of an array. Only the execution of the closure's body is deported to the hardware accelerator targeted by OpenCL:

Listing 4: Computing cosine of the 1000000 first integers through ScalaCL

```
import scalacl._
import scala.math._

implicit val context = new ScalaCLContext

val range = (0 to 1000000).cl
val rangeArray = r.toCLArray

// Runs asynchronously on the GPU via OpenCL
val mapResult = rangeArray.map ( v => cos(v).toFloat )
```

The obvious asset of ScalaCL is its ability to generate OpenCL at compile time. It means that we could enhance ScalaCL by adding an extra step to tune the issued OpenCL kernel at compile time and take advantage of a GPU vendor specific extensions for instance.

4.2.2 Aparapi

Aparapi [2] is a project initiated by AMD and recently freed under an MIT-like open source licence. It intends to provide a way to perform OpenCL actions directly from a pure Java source code. This process involves no upstream translation step, and is then wholly carried out at runtime. To do so, Aparapi relies on a Java Native Interface (JNI) wrapper of the OpenCL API that hides complexity to developers.

Basically, Aparapi proposes to implement the operations to be performed in parallel within a Kernel class. The kernel code takes place in an overridden *run()* abstract method of the Kernel class that sets the boundaries of the parallel computation. At the time of writing, only a subset of Java features are supported by Aparapi, it means that *run()* can only contain a restricted amount of features, data types, and mechanisms. Concretely, primitive data types, except char, are the sole data elements that can be manipulated within Aparapi's kernels.

For years, Java has been used to managing concurrency problems thanks to a Thread class whose implementation makes use of the native thread of the underlying Operating System where the Java Virtual Machine (JVM) runs. Thread implements the Runnable interface that only consists in providing a *void run()* method that will be called when the thread is launched. This mechanism is widely adopted in the Java world, and most of the frameworks offering an abstraction layer to Thread employ it in order to

remain compliant between each other. At first, it seems that Aparapi follows the same path given that it designates a *run()* method to contain the parallel code. However, Aparapi's source code does not involve Runnable at any moment. A simple example of using Aparapi is set up in Listing 5 to square an array of 8 integers:

Listing 5: Squaring an array of integers using Aparapi

```
final int[] inArray = new int[] {1, 2, 3, 4, 5, 6, 7, 8};
final int[] outArray = new float[input.length];

Kernel kernel = new Kernel ( ) {
        public void run() {
        outVector [ getGlobalId() ]  = inArray [ getGlobalId() ] *
                                       inArray [ getGlobalId() ];
        }
}

kernel.execute(inArray.length);
```

This design choice seems rather awkward when taking into consideration that the parallel tasks are assigned to a pool of CPU worker threads, the Java Thread Pool (JTP), when no accelerator can be found on the host platform. In fact, several implementations of a JTP are now shipping with Java Standard Development Kit (SDK) like Executors or Fork/Join, and have proved to be both efficient and user-friendly. Software design fosters reutilization as much as possible and being compliant with standard tools not only fastens development but it also strengthens it.

Additionally, it is interesting to note that Aparapi is not fully cross-platform, albeit being written in Java. The JNI bindings used to perform the calls to the OpenCL API make the Java package of Aparapi bound to native binaries. Thus, Aparapi cannot be shipped as a single package and depends on the ability of its underlying platform to run native code. This aspect can become a problem in terms of simplicity of use by clients, since it forces them to have a C++ tool-chain installed on each platform they want to run Aparapi on, so that the native part of the library can be recompiled for their particular platform. Still, native code dependency is not a major drawback for Aparapi given that there are few chances that a platform cannot execute the compiled-side of the JNI part of the library. This means that Aparapi can mostly be considered as a cross-platform tool that requires little effort from the client to be effective.

The first versions of Aparapi came with a slight limitation: AMD constrained the library to run on its devices only! Systems where AMD devices could not be found used to fall back to the JTP instead. Thanks to the open-source licence that protects this tool, a third-party developer was able to remove that latter constraint. In fact, execution is locked on a particular set of devices, here AMD GPUs, only by comparing the OpenCL platform identifier to the string identifying an AMD OpenCL platform. The removal of this software restriction makes Aparapi a potential high-level API to design HPC application.

4.3 A complete solution: JavaCL

JavaCL is an open source initiative that targets a smooth integration of OpenCL in Java applications. Like ScalaCL that we have studied in the previous section, JavaCL is part of the Nativelibs4java project developed by Olivier Chafik. The JavaCL library displays two interesting aspects that we will describe hereafter.

The first input from this library is to ease OpenCL development from the host side. In order to be easily integrated in Java code, the OpenCL 1.1 API [7] has been fully wrapped in Java classes. Every element required to perform OpenCL computations can now be handled directly in Java. This allows us to write nothing but the kernel using the C OpenCL API.

JavaCL is issued as a single *jar* file containing all the dependencies needed to be executed on any platform, included native libraries in charge of being the bridge between Java and the OpenCL platform. This way, it is perfectly independent from its underlying platform, and can be easily shipped by several ways. The most convenient one in our opinion is a Maven repository. Maven [3] is a build tool targeting Java applications that allows the projects it constructs to declare dependencies located in remote repositories. When a dependency is encountered for the first time, the maven client installed on the host system building the project downloads the dependency, and stores it in a local repository, so that it can be reused in a further build.

Functions intending to query an OpenCL installation have been designed so as to provide the most information in a single call. This completeness leads to an invocation pattern well-known by OpenCL developers: the double-call pattern foreseen in section 2.2. For instance, a function such as *clGetPlatformIDs* will return a list of the available OpenCL platforms in an array passed as a parameter along with its size. This array must have been allocated upstream and its size is consequently the maximum amount of results it can store. In addition to filling the array with the available platforms, *clGetPlatformIDs* also returns the total number of platforms in the system. Consequently, an application needs to invoke *clGetPlatformIDs* twice in order to figure out dynamically the amount of platforms on a given system: first, the function is called to determine the number of platforms and allocate an array according to this result, second, the function is summoned to actually fill the array. Such a design is widely used in the OpenCL API, and developers often have to call the same function twice in a row to make their code portable. The whole process results in bloated source files, whose real behaviour might become difficult to comprehend. JavaCL answers this awkwardness through its API that automatically issues a filled array as a result.

When platforms have been identified, it is time for an OpenCL application to list their belonging devices and to create a context combining some of them. As long as a high-performance computing application's main purpose is to speed up computation time, developers usually select the most efficient devices to form the OpenCL context in which their application will run. Once again, JavaCL takes care of this step by providing a *createBestContext* method that creates a context with the device containing the most work units.

The two previous features focused on increasing the ease of use of the OpenCL API. Moreover, JavaCL also enhances OpenCL development thanks to the Java language capabilities. Let us now examine the characteristics that make JavaCL really more than a simple OpenCL wrapper.

As we have seen in our introduction to OpenCL, the latter stores data that are to be treated on the device into buffers. The main concern with these buffers is that whatever the data they contain, they are represented by the sole *cl_mem* type. This is a potential source of harm since the compiler has no way to check that you are passing a wrong buffer type as parameter to a kernel for example. On the other side, object oriented languages such as Java are used to encapsulating data in dedicated containers. JavaCL does so by providing a different class for each data type that contains a buffer. Thanks to Java generics, a *CLBuffer* class can be parameterised with the primitive data type the buffer actually contains. Not only expressive containers is a nice feature for developers who have to correct a source code they did not write, but it is a particularly good point that helps compilers finding errors. Strong static typing prevents errors encountered when buffers from any type are accepted as parameters, which can appear dramatic when programming devices such as GPUs or FPGAs. To sum up, the JavaCL API allows the compiler to check little programming errors at compile time, before they turn into malicious bugs at runtime.

Being compiled at runtime, within the host program, OpenCL sources might lead to compiling errors as any program would. However, due to on-the-fly compiling, errors are a bit more tedious to take into account than in a usual program. Developers have to explicitly request for the compilation error log through the classical double call syntax inherent in the OpenCL API query system. This is not particularly suited to solve problems efficiently, and one has to make sure to correctly handle the output of the compiler anytime he builds a kernel through the C API. In the Java world, runtime errors are traditionally handled by exceptions. The exceptions mechanism is cleverly adapted by JavaCL so that

any kernel build will rise a *CLBuildException* that can be caught, and whose content can be printed easily.

In the same way, enqueued tasks errors are also wrapped in JavaCL exceptions. An exception hierarchy has been designed to cover the scope of potential errors issued by an OpenCL-enabled application. Explicit exceptions matching the most spread OpenCL errors can be casted throughout the application lifetime; for example, memory-bound errors corrupting the command queue are covered by *CLException.InvalidCommandQueue*. Once again, this JavaCL feature avoids the developer having to deal with involved errors codes, while the way Java handles exceptions ensures that errors will be reported at one time or another.

Last but not least, the most promising part in our opinion is the Generator. All the previous features introduced in this section either wrapped OpenCL calls or enhanced the API with type and error checking capabilities for example. The interesting point with the JavaCL generator is its ability to parse a pure OpenCL source file containing one or several kernels, and to issue a corresponding Java source file. The latter contains a class that associates a method per kernel present in the source file. In order to smoothly integrate in the host application, the automatically generated Java class is named after the source file name, whereas each of its methods bears the name of a kernel parsed in the file. Two aspects need to be distinguished to fully understand the inputs the generator brings to OpenCL development.

First, it widely contributes to simplifying OpenCL development: while JavaCL wrappers already make OpenCL on-the-fly compilation comfortable, the Generator allows developers to skip this step by reading the source code at compile time and generating a corresponding class whose methods can be called to launch the associated kernel.

Second, applications safety is again enforced given that kernel parameters are now typed. Traditionally, kernel parameters need to be set once and for all before enqueueing the kernel. JavaCL brings a first improvement to this process by allowing parameters to be all passed at once as a list of Objects. However, this behaviour might lead to runtime errors since the type and order in which parameters are passed cannot be verified at compile time, each of them being identified by a reference to an Object. Here, the Generator enables an earlier detection of this kind of error since the prototype of the method acting as a proxy of the kernel to be enqueued will only match the right type and order of the parameters. Listing 6 displays the execution of an instance of the kernel *MyKernel*, which corresponding class was generated upstream by the JavaCL Generator:

Listing 6: Squaring an array of integers using JavaCL

```
CLContext context = JavaCL.createBestContext();
CLQueue queue = context.createDefaultQueue();

final int vectorSize = 36864;
Pointer<Integer> inPtr = allocateInts (vectorSize);

for (int i = 0; i < vectorSize; ++i) {
   inPtr.set(i,i);
}

CLBuffer <Integer> inVector = context.createIntBuffer(Usage.Input,
                                                      inPtr);

// Create an OpenCL output buffer
CLBuffer<Integer> outVector = context.createIntBuffer(Usage.Output,
                                                      width*height*8);
```

```
// Read the program sources and compile them
MyKernel kernel = new MyKernel(context);

CLEvent addEvt = kernel.foo(queue, inVector, vectorSize,
                                        vectorSize / 192);

// Blocks until the kernel finishes
Pointer<Integer> outPtr = outVector.read(queue, addEvt);
```

Finally, the combination of an enhanced wrapping API and a generation mechanism proves that JavaCL goes further than the other APIs studied here.

4.4 Summary table of the solutions

In Tab. 1, we compare the APIs studied in this work according to the availability of three features: high-level wrapper, code generation facility, cross-platform portability of the resulting binary that will run on the host.

Table 1: Comparison of the studied APIs according to three criteria

API	Wrapper	Code-generator	Cross-platform
C++ Wrapper	Yes	No	No
QtOpenCL	Yes	No	No
PyOpenCL	Yes	Yes	Yes
ScalaCL	Yes	Yes	Yes
Aparapi	Yes	Yes	No
JavaCL	Yes	Yes	Yes

Tab. 1 states that although C++ is one of the most spread language in the HPC community, it suffers from poor APIs to abstract OpenCL.

5 Perspectives

Source-to-source solutions might lead to interesting works in the field of Model Driven Engineering (MDE). MDE is a modelling approach that emphasises two main aspects: the first are models and meta-models that allow, for instance, to represent Domain Specific Modelling Languages (DSML), and are generally expressed in the UML formalism. The second element guiding MDE are transformation engines and code generators, they bring a dynamic part in MDE. Their role is to define how meta-models are bound together, and how specific code can be generated from the information they contain. We observe at least a three-level hierarchy often presented as a pyramid, where meta-models are at the top level (M2), classical models at the intermediate one (M1), and the last level represents the real world (M0). In our case, we can consider both OpenCL as a meta-model of kernel functions. We can say that kernels are conform to OpenCL. On the other side, programming constructs such as VHDL, NVIDIA PTX or Intel SSE instructions are three equivalent kinds of models describing concrete binaries or RTL designs, i.e.: the real world. These languages obviously display specificities but their final purpose is the same, so their common aspects can be factored in a meta-model to which they conform. Let us call this meta-model Meta-Language. Given that OpenCL and Meta-Language are two meta-models, MDE allows us to transform elements of the same level automatically, provided valid transformations are available from

one meta-model to another. Thus, this modelling way would provide a base when enabling new targets to support source-to-source compilation of OpenCL applications, allowing developers to save lots of time and to simply follow the guidelines contained in the transformation rules.

Fig. 1 sums up the potential skeleton of an MDE approach to handle OpenCL source-to-source conversion to other languages:

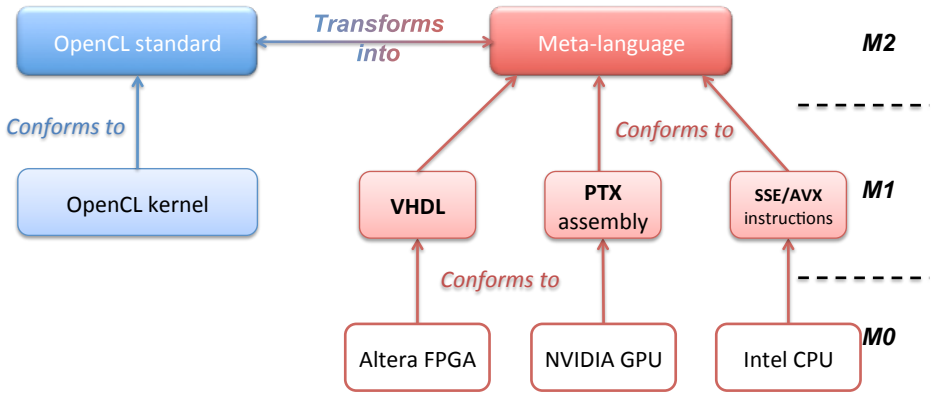

Figure 1: Skeleton of an MDE approach to transform OpenCL into any programming language

6 Conclusion

In this chapter we have attempted to show that the OpenCL standard was a great tool offering portability in high performance computing and authorizing a wider usage of GPUs, FPGAs, and other manycore platforms. We have seen that a raw usage of OpenCL does not meet the comfort of higher level APIs that parallel developers expect to help them avoid common mistakes, and to produce efficient applications in a reasonable amount of time. We have presented different approaches enabling software reuse and even more portability with features that avoid the developed code to be bound to the intrinsic characteristics of hardware platforms. Thus developers can base their developments upon reliable software components that have been tested many times and across several architectures. This consideration opens a new scope on the benefits of OpenCL for HPC. Using a cross-platform standard gives new testing perspectives. Indeed, such a feature obviously profits to heterogeneous computing approaches where FPGA can now be integrated among CPUs and GPUs using OpenCL. Not only do we have at our disposal abstraction levels much more understandable for developers not used to HDL-enabled codes, but we have seen in the previous sections that lots of APIs intend to bind OpenCL to widespread high-level languages such as C++ or Java. At the same time, legacy codes written in these languages can now benefit from the horsepower of manycores, without having to mix several programming languages in order to enable an application to exploit new hardware accelerators.

On the sole FPGA development side, OpenCL can also bring interesting features thanks to its ability to rely on the compiler, and the vendor driver to produce the hardware dependent part of the application. By doing so, migrations to a new FPGA versions do not rely on developers anymore but rather on the vendor's drivers. The involved application should consequently scale smoothly requiring no or few interventions from developers.

Last but not least, an abstraction layer that automatically generates low-level code changes the way FPGA applications are developed. Although several Intellectual Property (IP) cores are available for various purposes such as the support of Ethernet, they still need to be routed to be enabled. In the

same way, developers might spend a lot of valuable time to design state machines that will rule the computation on the FPGA. Albeit being important for a successful development, these considerations distract developers from the principal purpose of any HPC application: speed-up!

As a conclusion, delegating low-level tasks that can be automated to the backend of an abstraction layer should improve the quality of the parallel developments, since they will take up a larger place in the roadmap of an application.

Acknowledgements

The authors would like to thank D. S. Hill, Khaled Benkrid and Wim Vanderbauwhede for their careful reading and useful suggestions on the draft. This work was partly funded by the Auvergne Regional Council.

References

[1] Altera Corporation, "Implementing FPGA Design with the OpenCL Standard", Technical report, Altera Corporation, 2011.

[2] AMD, "Aparapi", `http://code.google.com/p/aparapi/`, 2011.

[3] Apache Software Foundation, "Apache Maven Project", `http://maven.apache.org/`, 2002.

[4] O. Chafik, "ScalaCL", `http://code.google.com/p/scalacl/`, 2011.

[5] S. Craven, P. Athanas, "Examining the viability of FPGA supercomputing", *EURASIP Journal on Embedded systems*, (1): 8, 2007, Research Article ID 93652.

[6] P. Jääskeläinen, C. de La Lama, P. Huerta, J. Takala, "OpenCL-based design methodology for application-specific processors", in *Embedded Computer Systems (SAMOS), 2010 International Conference on*, pages 223–230. IEEE, 2010.

[7] Khronos OpenCL Working Group, "The OpenCL Specification 1.2", Specification 1.2, Khronos Group, November 2011.

[8] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, A. Fasih, "PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation", *Parallel Computing*, 38(3): 157–174, 2012, ISSN 0167-8191.

[9] C. Lattner, V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation", in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[10] Nokia, "QtOpenCL", `http://doc.qt.nokia.com/opencl-snapshot/`, 2010.

[11] M. Owaida, N. Bellas, K. Daloukas, C. Antonopoulos, "Synthesis of platform architectures from opencl programs", in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 186–193. IEEE, 2011.

[12] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong, W. Hwu, "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs", in *Application Specific Processors, 2009. SASP'09. IEEE 7th Symposium on*, pages 35–42. Ieee, 2009.

[13] J. Passerat-Palmbach, J. Caux, P. Siregar, D. Hill, "Warp-Level Parallelism: Enabling Multiple Replications In Parallel on GPU", in *Proceedings of the European Simulation and Modeling Conference 2011*, pages 76–83, 2011, ISBN: 978-90-77381-66-3 *(best paper award)*.

[14] M. Scarpino, *OpenCL in Action*, Manning Publications, Shelter Island, NY, 2011.

[15] J. Stratton, S. Stone, W. Hwu, "MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs", *Languages and Compilers for Parallel Computing*, pages 16–30, 2008.

[16] J. Williams, A. George, J. Richardson, K. Gosrani, S. Suresh, "Computational density of fixed and reconfigurable multi-core devices for application acceleration", 2008.

UMR 6158 CNRS

RESEARCH CENTRE
LIMOS - UMR CNRS 6158
Campus des Cézeaux
Bâtiment ISIMA
BP 10125 - 63173 Aubière Cedex
France

Publisher
LIMOS - UMR CNRS 6158
Campus des Cézeaux
Bâtiment ISIMA
BP 10125 - 63173 Aubière Cedex
France
http://limos.isima.fr/