UNIVERSITÀ DEGLI STUDI DI PISA

Facoltà di Scienze Matematiche Fisiche e Naturali

Corso di Laurea in Magistrale in Informatica

Tesi di laurea

Data parallel patterns in Erlang/OpenCL

Relatore

Prof. Marco Danelutto

Candidato

Ugo Albanese

Anno Accademico 2014/2015

Alla mia famiglia

Acknowledgements

I would like to thank professor Marco Danelutto for the guidance and the outstanding support provided throughout every phase of the project. Also my infinite gratitude goes to my family and friends without whom everything would have been tougher if not impossible.

Contents

List of Figures

Chapter 1

Introduction

This thesis presents the Skel OCL library for the Erlang programming language [3][13] implementing data parallel patterns using OpenCL [6][17], to tap into Graphic Processing Units (GPU) tremendous parallel processing abilities.

In the following sections we introduce the motivation (1.1) for such a library, then we will continue describing the goals of the project (1.2) and stating the achievements of this thesis (1.3). Finally, the last section (1.4) outlines the rest of the thesis, summarizing the others chapters.

1.1 Background and Motivation

Over the past decade, we witnessed the rise of ubiquitous parallel hardware. Not only CPUs have nowadays multiple cores, but also GPUs, which have stopped being just specialized graphics processors and have also become very capable computing engines sporting a remarkable number of cores (currently in the range of hundreds to thousands).

Since GPUs must, owing to its execution model, execute multi-threaded code with threads operating on different data with the same code (this paradigm is called STMD, Single Thread Multiple Data), they are used as CPU co-processors to exploit data parallelism (definition in 2.1.4) in the computation at hand.

The industry standard programming framework to operate on GPU is OpenCL (see chapter 2). Its main competitor is NVIDIA's proprietary solution called CUDA [19]; in contrast to OpenCL, CUDA can only work in conjunction with NVIDIA hardware.

In recent years Erlang has seen a rapid increase in adoption thank to its built-in support for concurrency and distribution (among other very desirable features, see chapter 2). Declaredly, Erlang is ill suited for that class of problems for which iterative performance and high arithmetic intensity are prime requirements. This limitation can be overcome harnessing the processing power of GPUs using the OpenCL framework.

Erlang is also the one of the languages on which of the European Paraphrase project [10] is focusing its work. Citing from the project's website:
The ParaPhrase project aims to produce a new structured design and implementation process for heterogeneous parallel architectures, where developers exploit a variety of parallel patterns to develop component based applications that can be mapped to the available hardware resources, and which may then be dynamically re-mapped to meet application needs and hardware availability.

Therefore extending Erlang so to support GPU programming falls entirely within the scope of the project.

One of OpenCL original goals is to allow developers to write a single program that can run on a wide range of systems, from cell phones, to nodes in massive supercomputers. It manages to achieve this goal exposing the hardware instead of hiding it behind some user-friendly abstraction. Erlang programmers, being Erlang a functional language, are instead accustomed to very high level programming style. Dealing with all the intricacies of the specific hardware platform would be a very rude awakening from the elegance of functional programming.

It is then an imperative to hide most of the tedious aspects of OpenCL programming behind some façade that would make feel the Erlang programmer at home. An answer to this requirement can be found in the methodologies developed by the research on algorithmic skeletons.

The algorithmic skeleton concept has been introduced by Murray Cole with his PhD thesis in 1988 [14]. Algorithmic skeletons were described as pre-defined patters encapsulating the structure of a parallel computation that are provided to user as building blocks to be used to write applications. Each skeleton corresponded to a single parallelism exploitation pattern. Skeletons have been categorized in several classes [22]; we focus on the ones belonging to the class of data parallelism exploitation patterns. Many skeletons programming frameworks, developed by several academic groups, are already available targeting a range of diverse systems and programming languages (e.g. FastFlow [9], SkepPU [11][16], SkelCL [7], SkeTo [8])

1.2 Goals

The goal of the project developed in this thesis is to provide Erlang with an easy to use, high performance, data parallel skeleton library exploiting GPUs' processing power. The implementation is required to be vendor neutral, so OpenCL is the only possible choice. The skeletons must be easy to use, possibly be a nearly perfect drop-in replacement to standard Erlang functions with similar semantic (minimal disruption principle [15]).

1.3 Achievements

We developed a prototype Erlang library, using OpenCL, implementing the data parallel skeletons to accelerate computations on lists of Erlang floats. The user code for such computations is specified writing OpenCL kernels using the OpenCL C (see 2.1.5) programming language. The library has a very user-friendly API since it is modeled after Erlang's list module, with which every Erlang programmer is

familiar. The outcome of the tests (see chapter 4) shows, on one hand, a significant performance improvement over a pure Erlang implementation and, on the other hand, an acceptable performance loss over a C++/OpenCL reference implementation.

1.4 Thesis Outline

The rest of the thesis has the following structure:

- *Chapter 2* introduces and then give some details of, the technologies used in the project: OpenCL and Erlang. In particular it discuss the mechanism provided by Erlang to implement native functions in C that we used to implement the library.

- *Chapter 3* presents the Skel OCL library, starting from the architectural design, continuing with the implementation and ending with an example of its use.

- *Chapter 4* is concerned with the analysis of the experiments carried out to benchmark the library in order to validate its performance. After a description of the test platform, three different applications that use the skeletons provided by the library will be shown. The final paragraph presents and evaluates the result of the tests.

- *Chapter 5* presents the conclusions, recapping the achievements of the thesis and suggesting some directions for future work.

Chapter 2

Background Technologies

This chapter introduces the two technologies used in the project: OpenCL and Erlang. In first part it covers OpenCL, presenting the conceptual foundations of the standard and how NVIDIA implemented it. The second part concerns with Erlang. There is firstly a brief introduction of the platform and the language, then an introduction of Erlang's mechanism to implement native functions: the NIF API. We concentrate on this API as it is one of the main building blocks of the projects; NIFs are, in fact, the mechanism chosen to use OpenCL from Erlang.

2.1 OpenCL

OpenCL [6] is an industry standard framework for programming computers composed of a combination of CPUs, GPUs, and other processors. These so-called heterogeneous systems have become an important class of platforms, and OpenCL is the first industry standard that directly addresses these architectures. It is a framework for parallel programming and it includes a language, API, libraries and a runtime system to support software development.

OpenCL delivers high level of portability by exposing the underlying hardware structure: the programmer must explicitly define the platform, its context, and how work is scheduled onto different devices.

OpenCL goal is to provide a low-level hardware abstraction layer for portability so that high-level frameworks can be built on.

First released in 2008, the OpenCL specification version 2.0 is available since November 2013; to date, very few vendors released 2.0 implementation [12], in fact the most supported version of the standard is still version 1.1.

Let's now take a look at OpenCL conceptual foundations, namely the different models defined by the standard:

- *Platform model* defines a high-level description of the heterogeneous system.
- *Execution model* defines an abstract representation of how streams of instructions execute on the heterogeneous platform.
- *Memory model* defines the collection of memory regions used by OpenCL and how they interact during an OpenCL computation.
- *Programming models* defines the high-level abstractions a programmer uses when designing algorithms to implement an application.

2.1.1 Platform Model

The OpenCL *platform model* defines a high-level representation of any heterogeneous platform used with OpenCL.

An OpenCL platform always includes a single host; it interacts with the environment external to the OpenCL program, including I/O or interaction with the user.

The host is connected to one or more OpenCL *devices*. The device is where the kernels (a stream of instructions) execute, therefore an OpenCL device is referred to as a *compute device*: it can be a CPU, a GPU, a DSP, or any other processor provided by the hardware and supported by the vendor.

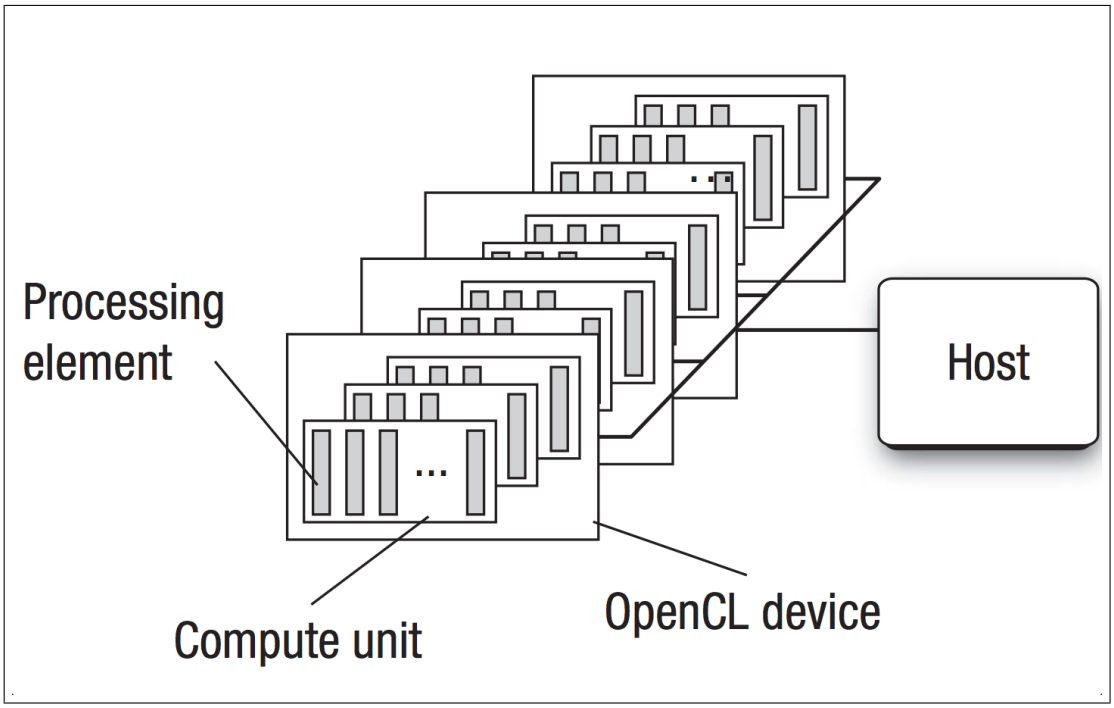

Figure 2.1: The OpenCL platform model with one host and one or more OpenCL devices. Each OpenCL device has one or more compute units, each of which has one or more processing elements.

The OpenCL devices are further divided into *compute units* and these are further divided into one or more *processing elements* (PEs). On-device computations occur within the PEs.

2.1.2 Execution Model

An OpenCL application is comprised of two distinct parts: the *host program* and a collection of one or more *kernels*. The host program runs, unsurprisingly, on the host. OpenCL does not define the details of how such program works, only how it interacts with objects defined within OpenCL.

The kernels execute on the OpenCL devices. Coded in the OpenCL C programming language and compiled with the OpenCL compiler, they do the actual work of an OpenCL application. Kernels are typically simple functions that transform input memory objects into output ones.

The OpenCL execution model defines how the kernels execute.

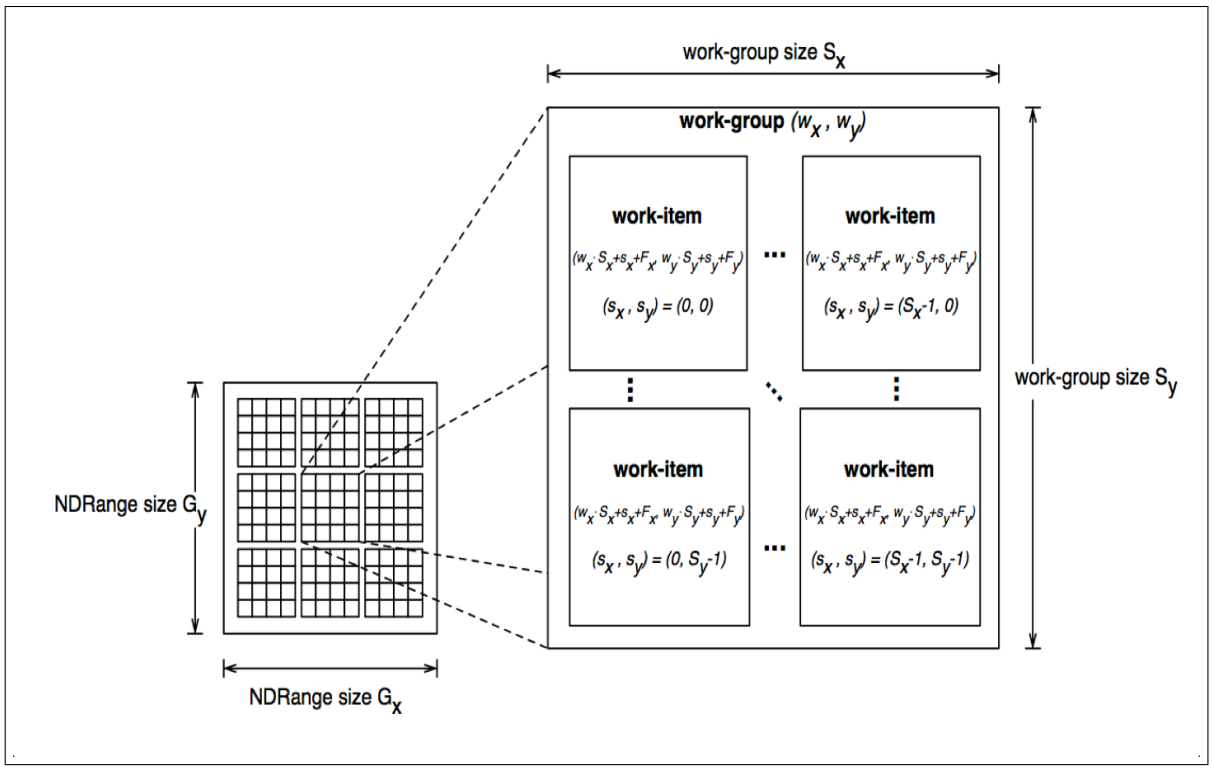

Figure 2.2: An example of an NDRange index space showing work-items, their global IDs and their mapping onto the pair of work-group and local IDs.

The host program issues a command that submits the kernel, which is defined on the host, for execution on an OpenCL device. When this command is issued by the host, the OpenCL runtime system creates an integer index space; an instance of the kernel executes for each point in this index space. Each instance of an executing kernel is called *work-item*, which is identified by its coordinates in the index space. These coordinates are the global ID of the work-item.

Although the sequence of instructions defined by a single kernel is the same for all work-items, the behavior of each one can vary because of branch statements within the code or data selected through the global ID.

Work-items are organized into *work-groups*. The work-groups provide a more coarse-

grained decomposition of the index space and exactly span the global index space. That is, work-groups are the same size in corresponding dimensions, and this size evenly divides the global size in each dimension. Work-groups are assigned a unique ID with the same dimensionality as the index space used for the work-items. Work-items are assigned also a local ID; therefore they can be uniquely identified either by its global ID or by a combination of its local ID and work-group ID.

The index space spans an N-dimensioned range of values and thus is called an NDRange. The N in this N-dimensional index space can, currently, be 1, 2, or 3. A NDRange is defined by an integer array of length N specifying the size of the index space in each dimension. Each work-item's global and local ID is an N-dimensional tuple.

Work-groups are dispatched to compute units so to be executed by its processing elements.

Context and command queues.

The host defines a context for the execution of the kernels that includes the following resources:

- *Devices*: The collection of OpenCL devices to be used by the host.
- *Kernels*: The OpenCL functions that run on OpenCL devices.
- *Program Objects*: The program source and executable that implement the kernels.
- *Memory Objects*: A set of memory objects visible to the host and the OpenCL devices. Memory objects contain values that can be operated on by instances of a kernel.

The context is created and manipulated by the host using functions from the OpenCL API. The host creates a data structure called *command-queue* to coordinate execution

of the kernels on the devices. The host places commands into the command-queue, which are then scheduled onto the devices within the context. The kinds of commands available are:

- *Kernel execution commands*: Execute a kernel on the processing elements of a device.
- *Memory commands*: Transfer data to, from, or between memory objects, or map and unmap memory objects from the host address space.
- *Synchronization commands*: Constrain the order of execution of commands.

The command-queue schedules commands for execution on a device. These execute asynchronously between the host and the device. The relative order in which commands execute depends on what mode are used for the queue: beginitemize

- *In-order Execution*: Commands are launched in the order they appear in the command-queue and complete in order. In other words, a preceding command on the queue completes before the following command begins. That establishes a serial execution order of commands in a queue.
- *Out-of-order Execution*: Commands are issued in order, but do not wait to complete before following commands execute. The programmer enforces any order constraints through explicit synchronization commands.

Kernel execution and memory commands submitted to a queue generate event objects. These are used to control execution between commands and to coordinate execution between the host and devices.

It is possible to associate multiple queues with a single context that will run concurrently and independently. We used this feature to optimize the performance of the library. In particular, we used two independent queues to concurrently submit commands to the GPU (see 3.2 for details).

2.1.3 Memory model

The execution model tells us how the kernels execute, how they interact with the host, and how they interact with other kernels. The Memory model, instead, define the details of memory objects and the rules to safely use them.

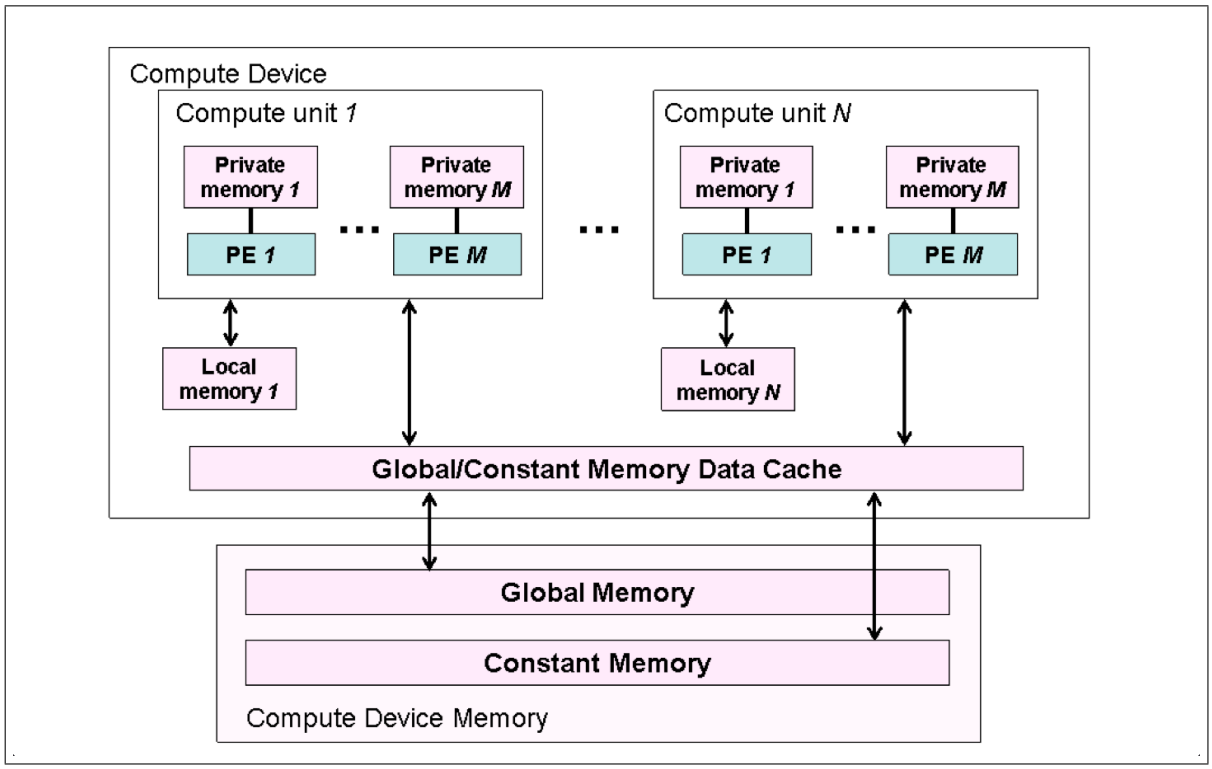

Figure 2.3: The OpenCL conceptual device architecture with processing elements (PE), compute units and devices. The host is not shown.

The OpenCL memory model defines five distinct memory regions:

- *Host memory*: It's visible only to the host. OpenCL defines only how the host memory interacts with OpenCL objects and constructs.
- *Global memory*: It permits read/write access to all work-items in all work-groups. Work-items can read from or write to any element of a memory object in global memory. Reads and writes to global memory may be cached depending on the capabilities of the device.
- *Constant memory*: This memory region of global memory remains constant during the execution of a kernel. The host allocates and initializes memory

objects placed into constant memory. Work-items have read-only access to these objects.

- *Local memory*: This memory region is local to a work-group. This memory region can be used to allocate variables that are shared by all work-items in that work-group. It may be implemented as dedicated regions of memory on the OpenCL device. Alternatively, the local memory region may be mapped onto sections of the global memory.

- *Private memory*: This region of memory is private to a work-item. Variables defined in one work-item's private memory are not visible to other work-items.

These memory regions form a hierarchy as shown in figure 2.3. Memory accesses of a work-item are increasingly slower as it request data from the upper levels, with the global one being the slowest.

OpenCL uses a relaxed consistency memory model: the state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times.

Within a work-item memory has load/store consistency. Local memory is consistent across work-items in a single work-group at a work-group barrier. Global memory is consistent across work-items in a single work-group at a work-group barrier, but there are no guarantees of memory consistency between different work-groups executing a kernel.

Memory consistency for memory objects shared between enqueued commands is enforced at a synchronization point.

2.1.4 Programming model

The OpenCL execution model supports data parallel and task parallel programming models, as well as supporting hybrids of these two models. Ultimately, what really drives the design of OpenCL is the data parallel one.

In *data-parallel* programming the problem is described in terms of collections of data elements that can be updated in parallel. The parallelism is expressed by concurrently applying the same stream of instructions to each data element: the parallelism is in the data. The index space associated with the OpenCL execution model defines the work-items and how the data maps onto the work-items. In a strictly data parallel model, there is a one-to-one mapping between the work-item and the element in a memory object over which a kernel can be executed in parallel. OpenCL implements a relaxed version of the data parallel programming model where a strict one-to-one mapping is not a requirement.

In particular, OpenCL provides a hierarchical data parallel programming model: there are two ways to specify the hierarchical subdivision. In the *explicit* model a programmer defines the total number of work-items to execute in parallel and also how the work-items are divided among work-groups. In the *implicit* model, a programmer specifies only the total number of work-items to execute in parallel, and the division into work-groups is managed by the OpenCL implementation.

In a *task-parallel* programming model, programmers directly define and manipulate parallel tasks. Problems are decomposed into tasks that can run concurrently, which are then mapped onto processing elements (PEs) of a parallel computer for execution. OpenCL supports task parallel programming defining a model in which a single instance of a kernel is executed independent of any index space. It is logically equivalent to executing a kernel on a compute unit with a work-group containing a single work-item. Under this model, users express parallelism by using vector data types implemented by the device or enqueuing multiple tasks.

2.1.5 The OpenCL C language

OpenCL kernels are written using the OpenCL C language. It is based on the ISO/IEC 9899:1999 C language specification (referred to in short as C99) with some restrictions and specific extensions to the language for parallelism. stantard, with some extensions.

The following C99 features (among others) are not supported:

- Function pointers
- Recursion
- Bit field struct members are not supported
- Variable-length arrays and structures with flexible (or unsized) arrays are not supported

Instead, OpenCL adds the following features to C99:

- *Vector data types.* In OpenCL C, vector data types can be used in the same way scalar types are used in C. This makes it much easier for developers to write vector code because similar operators can be used for both vector and scalar data types. It also makes it easy to write portable vector code because the OpenCL compiler is now responsible for mapping the vector operations in OpenCL C to the appropriate vector ISA for a device. Vectorizing code also helps improve memory bandwidth because of regular memory accesses and better coalescing of these memory accesses.
- *Address space qualifiers.* OpenCL devices such as GPUs implement a memory hierarchy. The address space qualifiers are used to identify a specific memory region in the hierarchy.
- *Additions for parallelism*: These include support for work-items, work-groups, and synchronization between work-items in a work-group.
- *An extensive set of functions* such as math, integer, geometric, functions.

2.1.6 NVIDIA OpenCL implementation

Being the platform on which the project has been developed and tested, equipped with a NVIDIA GPU, let's have a look at NIVIDIA OpenCL implementation.

To date, NVIDIA supports OpenCL up to version 1.1, lagging behind others vendors such as AMD.

NVIDIA's unenthusiastic support to OpenCL is due to the fact that they market their own proprietary standard for GPU computing, named CUDA [19], which is OpenCL's main competitor.

NVIDIA's OpenCL implementation is, therefore, based on CUDA: e.g. Kernels written in OpenCL C are compiled into PTX, which is CUDA's instruction set.

Let's see in more detail how CUDA architecture maps onto OpenCL one.

A CUDA device is built around a scalable array of multithreaded Streaming Multiprocessors (SMs). A multiprocessor corresponds to an OpenCL compute unit.

An SM executes a CUDA thread for each OpenCL work-item and a thread block for each OpenCL work-group. A kernel is executed over an OpenCL NDRange by a grid of thread blocks.

When an OpenCL program on the host invokes a kernel, the work-groups are enumerated and distributed as thread blocks to the multiprocessors with available execution capacity. The threads of a thread block execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors.

A multiprocessor is designed to execute hundreds of threads concurrently. To manage such a large amount of threads, it employs an architecture called SIMT (Single-Instruction, Multiple-Thread).

To maximize utilization of its functional units, it leverages thread-level parallelism by using hardware multithreading, more so than instruction-level parallelism within a single thread (instructions are pipelined, but unlike CPU cores they are executed in order and there is no branch prediction and no speculative execution).

The SM creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*. Individual threads composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently.

When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps that get scheduled by a warp scheduler for execution.

A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths.

The execution context (program counters, registers, etc.) for each warp processed by an SM is maintained on-chip during the entire lifetime of the warp. Switching from one execution context to another has therefore no cost, and at every instruction issue time, the warp scheduler selects a warp that has threads ready to execute (active threads) and issues the next instruction to those threads.

2.2 Erlang

2.2.1 Introduction

Erlang [3] is a functional general-purpose programming language and runtime environment developed by Ericsson since the 1980s; it has built-in support for concurrency, distribution and fault tolerance.

Since its open source release, Erlang has been adopted by many leading telecom and IT companies. Nowadays it is successfully being used in other industries including banking, finance and e-commerce.

Erlang is distributed with a large collection of libraries called OTP (Open Telecom platform) [4] allowing the user to easily develop applications using anything from telecommunication protocols to HTTP servers. It also provides implementations of several patterns that have proven useful in massively concurrent development over the years. Most production Erlang applications are actually Erlang/OTP applications. OTP is also open source and distributed with Erlang.

Although Erlang is a general-purpose language, it is well suited especially for the application domain for which it was designed. This is a niche mostly consisting of distributed, reliable, soft real-time concurrent systems. These types of applications are telecommunication systems, Servers and Database applications, which require soft real-time behavior.

However, the most common class of problems where Erlang is not indicated is characterized by iterative performance being a prime requirement. To such class belong applications like image processing, signal processing and sorting large volumes of data.

It is evident how much Erlang could benefit from GPU raw processing power in solving the very same class of problems for which it is, currently, ill suited.

2.2.2 Erlang Features

Let's now have a look at Erlang's core features:

- *Concurrency* - A process in Erlang is the entity that carries out work. They are fast to create, suspend or terminate being much more lightweight than OS ones. A running Erlang system may even have millions of concurrent processes, each of them having its own memory area, which is not shared with the other processes. The memory each process allocates can change dynamically during its execution upon request. Processes communicate only by asynchronous message passing.The sending of messages is non-blocking; the receiving, however, suspends the process until a matching message is delivered to its mailbox, i.e. a message queue.

- *Distribution* - Erlang is designed, from the ground up, to run in a distributed environment. An Erlang VM is called a node and a distributed Erlang system is just a set of networked Erlang nodes. Every node can create parallel processes running on other nodes in other machines independently form their operating system. Communication between processes residing on different nodes uses the same mechanism used by processes on the same node to communicate.

- *Robustness* - *Erlang* supports an exception detection and recovery mechanism based on the catch/throw-style. It also offers a supervision infrastructure: a process can register to be the supervisor of another, and receive a notification message if the supervised process terminates. The node under supervision can even reside in a different machine; once notified the supervisor can implement its preferred recovery policy.

- *Hot code replacement* - The high availability needed by Erlang's original domain of application, telecom systems, demanded the system not to be halted even during software updates. Thus, Erlang provides a way of replacing running code without stopping the system: the runtime system maintains a global table containing the addresses for all the loaded modules. These addresses are updated whenever new modules replace old ones. Future calls invoke functions

in the new modules. It is also possible for two versions of a module to run simultaneously in a system.

- *Soft real-time* - Erlang supports developing soft real-time applications with response time demands in the order of milliseconds.
- *Memory management* - Memory is managed by the VM automatically using garbage collection techniques. When a process terminates, its memory is simply reclaimed.

Moving to the more pragmatic aspects of the language, Erlang is a dynamically typed language. Variables don't need to be declared before their use, they are single assigned: once bound to a value they cannot be reassigned. Like in the vast majority of functional languages, values are immutable.

Erlang's basic data types are number, atom, function type, binary, reference, process identifier, and port identifier.

- *Atoms* are constant literals, and are similar to the enumerations used in other programming languages.
- *Functions* are first class objects of the language: they are a data type, can be passed as an argument to other functions, or can be a returning result of a function.
- *Binaries* are a reference to an area of raw memory so they are used as an efficient way for storing and transferring large amounts of data.
- *References* are unique values generated on a node for the purpose of identifying entities.
- *Process* and *port identifiers* are used to uniquely identify different processes and ports. Ports are used to pass binary messages between Erlang nodes and external programs possibly written in other programming languages and residing in another OS process. A port is created and owned by an Erlang process, which is responsible for coordinating all the messages passing through that port.

The native (implemented in C) counterpart of an Erlang function is the so-called NIF (*Natively implemented Function*). The NIF mechanism will be analyzed in the next section, being one of the cornerstones of the project.

Besides its basic data types, Erlang provides the classical functional data structures such as tuples and lists, adding to that also records.

Tuples and lists are collections of any valid Erlang data-type. From a tuple we can only extract a particular element, but a list can be split and combined. Records in Erlang are akin to C structures, having a fixed number of named fields.

The basic units of code in Erlang are *Modules*. Every module has a number of functions that can be called from other modules if the programmer exports them.

Functions are, instead, the basic unit of abstraction; they consist of several clauses. The right clause to be evaluated is chosen at runtime by pattern matching the calling arguments against the specified pattern.

Erlang doesn't provide constructs for looping; recursion is used instead. To reduce stack consumption, tail call optimization is implemented.

2.2.3 Native Implemented Functions (NIF)

Erlang, from version R14B onwards, provides an API to implement native functions, in C/C++, that are called in the same way as the Erlang implemented ones.

Such functions are called NIF (Native Implemented Functions) and, to interact with Erlang's VM, they use the API provided in the `erl_nif` module [2].

A NIF library contains native implementation of some functions of an Erlang module. Each NIF must also have an implementation in Erlang that will be invoked if the function is called before the NIF library has been successfully loaded, so to throw

an exception or, perhaps, provide a fallback implementation if the NIF library is not implemented for some architecture.

Since a native function is executed as a direct extension of the native code of the VM, its execution is not made in a safe environment. That is, if the native function misbehaves, so the whole VM will. In particular, quoting the module's documentation [2]:

- *A native function that crashes will crash the whole VM*
- *An erroneously implemented native function might cause a VM internal state inconsistency, which may cause a crash of the VM, or miscellaneous misbehaviors of the VM at any point after the call to the native function.*
- *A native function that does lengthy work (more than 1ms long) before returning will degrade responsiveness of the VM, and may cause miscellaneous strange behaviors. Such strange behaviors include, but are not limited to, extreme memory usage, and bad load balancing between schedulers. Strange behaviors that might occur due to lengthy work may also vary between OTP releases.*

As workarounds for the last limitation, several methods are suggested in the documentation, depending on the ability to fully control the code to execute in the native function. If that is the case, the best approach is to divide the work into multiple chunks of work and call the native function multiple times; a function is provided for helping with such work division.

If full control of the code is not possible (e.g. calling third-party libraries) then is recommended to dispatch the work to another thread, return from the native function, and wait for the result. The thread can send the result back to the calling thread using message passing.

As another way to solve the problem, in version R17B a new kind of schedulers have been introduced for dealing with long running functions.

A "dirty NIF" is a NIF that cannot be split and cannot execute in a millisecond or less so it performs work that the Erlang runtime cannot handle cleanly. Building the system with the currently experimental support for dirty schedulers, and dispatching the work to a dirty NIF, the user can ignore the problem altogether since such schedulers will take care of everything.

Although the usage of dirty NIFs is generally discouraged, for some applications it may be considered acceptable. For example, if the only work done by an Erlang program is carried out by the NIF, scheduling interferences are not a problem.

NIF API Functionalities

Every interaction between a NIF library and Erlang, is performed through the NIF API functions.

Several functions are provided for the following functionalities:

Handling of Erlang terms
Any Erlang terms can be passed to a NIF as function arguments and be returned as function return values. The terms can only be read, written and queried using API functions.
All terms belong to an environment. The lifetime of a term is controlled by the lifetime of its environment object. All API functions that read or write terms have the environment that the term belongs to, as the first function argument. There are two types of environments:

- *A process bound environment* is passed as the first argument to all NIFs. All function arguments passed to a NIF and its return value will belong to that environment. The environment is only valid in the thread where it was supplied as argument until the NIF returns. It is not only useless but also dangerous to store pointers to process bound environments between NIF calls.

- *A process independent environment* is user created and can be used to store

terms between NIF calls and to send term. A process independent environment with all its terms is valid the user explicitly invalidates it.

Terms can be copied between environments `with enif_make_copy`.

Binaries

Terms of type binary, like every other Erlang term, must be handled only through calls to API functions. Instances of them, however, are always allocated by the user.

A mutable binary must in the end either be freed `with enif_release_binary` or made read-only by transferring it to an Erlang term `with enif_make_binary`. Read-only binaries do not have to be released.

Resource objects

Extensively used in Skel OCL to allow OpenCL objects handling in Erlang programs (see Chapter 3), Resource objects (ROs) allows for a safe way to return pointers to native data structures from a NIF. A resource object is just a block of memory allocated by the user. A handle ("safe pointer") to this memory block can then be returned to Erlang, but it is totally opaque in nature. It can be stored and passed between processes on the same node, but the only real end usage is to pass it back as an argument to a NIF. The NIF can then get back a pointer to the memory block that is guaranteed to still be valid. A resource object will not be deallocated until the VM has garbage collected the last handle term and the user has released the resource (in any order). All resource objects are created as instances of some resource type: this makes resources from different modules to be distinguishable.

Threads and concurrency

A NIF is thread-safe without any explicit synchronization as long as it acts as a pure function and only reads the supplied arguments. As soon as it writes towards a shared state, it must supply its own explicit synchronization. This includes terms in process independent environments that are shared between. Resource objects will also require synchronization if treated as mutable.

Chapter 3

The Skel OCL library

Skel OCL is a prototype for an Erlang data parallel skeleton library that exploits GPUs' processing power using the OpenCL programming framework.

Skel OCL allows the Erlang programmer to use map and reduce data parallel skeletons on lists of Erlang floats. The function to apply to the elements of the list is specified as OpenCL kernel, written in OpenCL C (see 2.1.5).

A lot of care has been taken to make the use of the library as friendly as possible to the Erlang programmer. In fact, as we will see in the examples in the next chapter, map and reduce skeletonds are nearly perfect drop-in replacement for the functionally equivalent counterparts contained in Erlang's list module.

In addition to the skeleton NIFs, Skel OCL provides a set of low-level functions that make possible to compose the skeletons at will. Using these functions the user can even define new skeletons; in paragraph 3.4 we will show how easily this can be done.

In this chapter we firstly explain Skel OCL Architectural Design (and give an example of usage), secondly we look at the implementation details. Finally we show how to define, using the functions provided by the library, a new data parallel skeleton, namely the `MapReduceLL skeleton`.

3.1 Architectural Design

Skel OCL is implemented as Erlang NIF library (see 2.3 for an intro to NIFs) where the native implementation of the functions is written in C++; to work with GPUs, these functions uses OpenCL. A components diagram for the library is shown in figure 3.1.

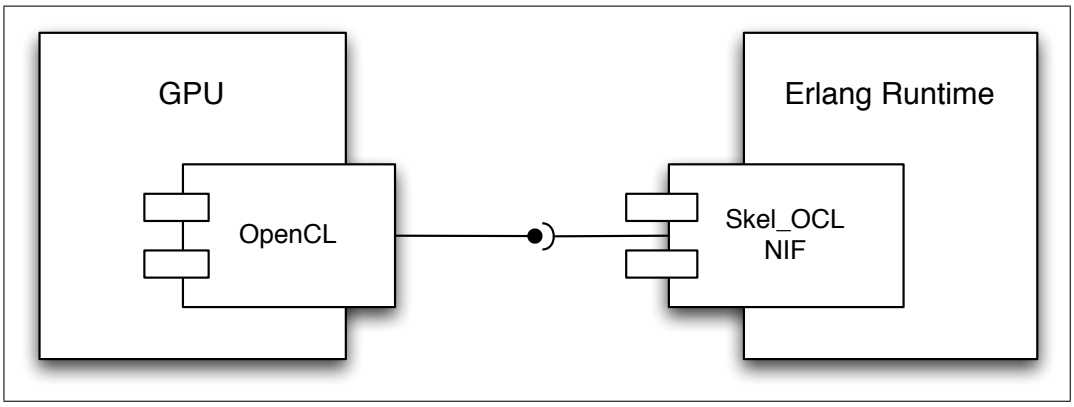

Figure 3.1: Skel OCL components diagram.

A NIF library is a set of natively implemented functions (in C/C++) compiled in a shared library that are loaded at runtime by the Erlang VM. As specified in the documentation [2], each native function must have an Erlang counterpart. These Erlang functions are used as fallback implementation in case the loading of the NIF fails (see also 2.2.3).

In the following sections we details one by one the main components of the library.

3.1.1 The library API: `skel_ocl.erl`

The Erlang interface of the library is defined in the `skel_ocl.erl` file, in which the `skel_ocl` module is contained.

OpenCL bindings

Since OpenCL is used for interacting with GPUs, Skel_OCL provides a set of NIFs to handle OpenCL objects like buffers, programs and kernels (see 2.1 for details on OpenCL). These objects are implemented using the Resource Objects mechanism

(see 2.2.3).

Let's have a look at these NIFs.

- *Program and Kernel NIFs* – In OpenCL the functions are called "kernels", they run on OpenCL devices and are written in OpenCL C; the program source and executable that implement the kernels are called "program objects". A program object is like a dynamic library in that it contains a collection of kernel functions. A kernel object instead is like a handle to a function within the dynamic library. Skel OCL provides functions for creating program and kernel objects. Program objects are created reading the code directly from a specified source file. Kernels objects are, instead, created from program objects (same as OpenCL). The specification for these NIFs is shown in figure 3.2

```
%%Program and Kernel Types and functions ---------------------------------

-type program()          ::      binary().
-type kernel()           ::      binary().

%%Build an OpenCL program from the source code in the ProgSrcString iolist()
-spec  buildProgramFromString(ProgSrcString) -> {ok, program()} | {error, Why} when
        ProgSrcString    ::      iolist(),
        Why              ::      atom().

%%Build an OpenCL program object from the source code in the ProgSrcFile file
-spec  buildProgramFromFile(ProgSrcFile) -> {ok, program()} | {error, Why} when
        ProgSrcFile      ::      nonempty_string(),
        Why              ::      atom().

%%Create an OpenCL kernel object from a program object.
-spec  createKernel(Prog, KerName) -> {ok, kernel()} | {error, Why} when
        Prog             ::      program(),
        KerName          ::      nonempty_string(),
        Why              ::      atom().
```

Figure 3.2: Program and Kernel objects NIFs

- *Buffer NIFs* - OpenCL allows the user to allocate memory buffer either on the host memory or the device memory. This distinction is crucial since the cost of accessing the memory on the device is not negligible, moreover, in order to be used on the host it must be copied first.

Taking that into account, Skel OCL defines different NIFs for allocating host buffers and device buffers. In addition to these functions, NIFs for copying floats

from and to an Erlang list are provided. The functions' specification is shown in figure 3.3

```
%%Buffer Types and functions ---------------------------------

-type hostBuffer()      ::      binary().
-type deviceBuffer()    ::      binary().
-type buffer()          ::      hostBuffer() | deviceBuffer().

-type rw_flag()         ::      read | write | read_write.

%%Get the size of Buffer in Bytes
-spec getBufferSize(Buffer) -> {ok, SizeByte} | {error, Why} when
                SizeByte        ::      non_neg_integer(),
                Buffer          ::      buffer(),
                Why             ::      atom().

%%Allocate a buffer on the host of size SizeByte bytes
-spec allocHostBuffer(SizeByte) -> {ok, Buffer} | {error, Why} when
                SizeByte        ::      non_neg_integer(),
                Buffer          ::      hostBuffer(),
                Why             ::      atom().

%%Allocate a buffer on the device of size SizeByte bytes, specifying kernel read/write permissions
-spec allocDeviceBuffer(SizeByte, Flags) -> {ok, Buffer} | {error, Why} when
                SizeByte        ::      non_neg_integer(),
                Flags           ::      rw_flag(),
                Buffer          ::      deviceBuffer(),
                Why             ::      atom().

%%release a previously allocated buffer
-spec releaseBuffer(Buffer) -> ok when
                Buffer          ::      buffer().

%%Copy data from a buffer to another having the same size
-spec copyBufferToBuffer(From, To) -> ok | {error, Why} when
                From            ::      buffer(),
                To              ::      buffer(),
                Why             ::      atom().

%%Copy CopySizeByte data from a buffer to another
-spec copyBufferToBuffer(From, To, CopySizeByte) -> ok | {error, Why} when
                From            ::      buffer(),
                To              ::      buffer(),
                CopySizeByte    ::      non_neg_integer(),
                Why             ::      atom().

%%Copy the content of an Erlang list into a host buffer.
-spec listToBuffer(From, To) -> ok | {error, Why} when
                From            ::      [float()],
                To              ::      hostBuffer(),
                Why             ::      atom().


%%Create a list having as elements the data in the host buffer
-spec bufferToList(From) -> {ok, [float()]} | {error, Why} when
                From            ::      hostBuffer(),
                Why             ::      atom().
```

Figure 3.3: Buffer management NIFs

Skeletons NIFs

We now introduce the higher-level NIFs of the library: the *Map and Reduce* Skeleton NIFs. Skeleton NIF functions are named according to the following pattern: to the name of the skeleton two capital letters are appended, the first is the type of the input and the second is the one of the output. The possible types are `List` and `Device buffer`. For example `mapLD` is a map skeleton that takes lists and outputs the result into a device buffer, `while reduceDL` is a reduce skeleton from device buffers to lists.

Skeletons are generic components that can be parameterized with actual computations in the form of user-functions. In Skel OCL such functions are specified in OpenCL C.

NIFs to generate and compile the OpenCL programs are available, one for each type of skeleton. The kernel object returned by these NIFs, is used as functional parameter of the skeletons (see chapter 4 for examples of use).

- *Map* - The map skeleton is functionally equivalent to Erlang's `lists:map` [1] function (see the examples in chapter 4 for a comparison). A user specified function is applied to every element of the input list/buffer and the result of this function is appended to the output list/buffer. In addition to the classical map function, also a binary map (called map2) is provided, this skeleton's Erlang counterpart is `lists:zipWith` (see dotProduct example in chapter 4). The user-provided binary function is applied to the corresponding elements of the two input lists/buffers and the resulting value is concatenated to the output list/buffer. The specifications for the map NIFs are in figure 3.4.

- *Reduce* – The reduce skeleton is equivalent to the `lists:fold` family of Erlang functions. The fold function, along with map, is the fundamental tool of the functional programmer. The reduce function builds up a return value recombining, through use of a given combining operation, the results of processing its constituent parts in a recursive fashion. The specification for the reduce NIFs are in figure 3.6

```
%%Map functions ----------------------------

%%Create a map compatible kernel object from the source in MapSrcFunFile. The kernel name is FunName.
-spec createMapKernel(MapSrcFunFile, FunName) -> {ok, kernel()} | {error, Why} when
                MapSrcFunFile   ::      nonempty_string(),
                FunName         ::      nonempty_string(),
                Why             ::      atom().

%%Create a map compatible kernel object from the source in MapSrcFunFile. The kernel name is FunName, the
arity is FunArity
-spec createMapKernel(MapSrcFunFile, FunName, FunArity) -> {ok, kernel()} | {error, Why} when
                MapSrcFunFile           nonempty_string(),
                FunName         ::      nonempty_string(),
                FunArity        ::      pos_integer(),
                Why             ::      atom().

%%Create a map compatible kernel object from the source in MapSrcFunFile.
%%The kernel name is FunName, the arity is FunArity.
%%Program caching policy must also be specified.
-spec createMapKernel(MapSrcFunFile, FunName, FunArity, CacheKernel) -> {ok, kernel()} | {error, Why} when
                MapSrcFunFile   ::      nonempty_string(),
                FunName         ::      nonempty_string(),
                FunArity        ::      pos_integer(),
                CacheKernel     ::      cache | no_cache,
                Why             ::      atom().

%%Map skeleton working on device buffers. User-specified function specified as kernel objects.
-spec mapDD(Kernel, Input, Output) -> ok | {error, Why} when
                Kernel          ::      kernel(),
                Input           ::      deviceBuffer(),
                Output          ::      deviceBuffer(),
                Why             ::      atom().

%% Map skeleton: input is a list, output too. User-specified function specified as kernel objects.
-spec mapLL(Kernel, Input, InputLength) -> {ok, Result} when
                Kernel          ::      kernel(),
                Input           ::      [float()],
                InputLength     ::      non_neg_integer(),
                Result          ::      [float()].

%%Map skeleton: input is a list, output is a device buffer. User-specified function specified as kernel objects.
-spec mapLD(Kernel, Input, Output, InputLength) -> ok | {error, Why} when
                Kernel          ::      kernel(),
                Input           ::      [float()],
                Output          ::      deviceBuffer(),
                InputLength     ::      non_neg_integer(),
                Why             ::      atom().

%%Binary Map skeleton working on device buffers. User-specified function specified as kernel objects.
%%Similar to lists:zipWith
-spec map2DD(Kernel, Input1, Input2, Output) -> ok | {error, Why} when
                Kernel          ::      kernel(),
                Input1          ::      deviceBuffer(),
                Input2          ::      deviceBuffer(),
                Output          ::      deviceBuffer(),
                Why             ::      atom().
```

Figure 3.4: Map Skeleton NIFs

```
%%Binary Map skeleton: input is a list, output is a device buffer. User-specified function specified as kernel
objects.
-spec map2LD(Kernel, Input1, Input2, Output, InputLength) -> ok | {error, Why} when
                Kernel          ::      kernel(),
                Input1          ::      [float()],
                Input2          ::      [float()],
                Output          ::      deviceBuffer(),
                InputLength     ::      non_neg_integer(),
                Why             ::      atom().

%%Binary Map skeleton: input is a list, output too. User-specified function specified as kernel objects.
-spec map2LL(Kernel, Input1, Input2, InputLength) -> {ok, Result} when
                Kernel          ::      kernel(),
                Input1          ::      [float()],
                Input2          ::      [float()],
                InputLength     ::      non_neg_integer(),
                Result          ::      [float()].
```

Figure 3.5: Map Skeleton NIFs (continued)

```
%%Reduce functions ----------------------------

%%Create a reduce compatible kernel object from the source in ReduceSrcFunFile.
%%The kernel name is FunName.
-spec createReduceKernel(ReduceSrcFunFile, FunName ) -> {ok, kernel()} | {error, Why} when
                ReduceSrcFunFile        ::      nonempty_string(),
                FunName                 ::      nonempty_string(),
                Why                     ::      atom().

%%Create a reduce compatible kernel object from the source in ReduceSrcFunFile.
%%The kernel name is FunName
%%Program caching policy must also be specified.
-spec createReduceKernel(ReduceSrcFunFile, FunName, CacheKernel) -> {ok, kernel()} | {error, Why} when
                ReduceSrcFunFile        ::      nonempty_string(),
                FunName                 ::      nonempty_string(),
                CacheKernel             ::      cache | no_cache,
                Why                     ::      atom().

 %%Reduce skeleton working on device buffers. User-specified function specified as kernel objects.
-spec reduceDD(Kernel, Input, Output) -> ok | {error, Why} when
                Kernel          ::      kernel(),
                Input           ::      deviceBuffer(),
                Output          ::      deviceBuffer(),
                Why             ::      atom().

%%Reduce skeleton: input is a device buffer, output is a list. User-specified function specified as kernel objects.
-spec reduceDL(Kernel, Input) -> {ok, Result} | {error, Why} when
                Kernel          ::      kernel(),
                Input           ::      deviceBuffer(),
                Result          ::      [float()],
                Why             ::      atom().

%%Reduce skeleton: both input and output are lists. User function specified as kernel objects.
-spec reduceLL(Kernel, Input, InputLength) -> {ok, Result} | {error, Why} when
                Kernel          ::      kernel(),
                Input           ::      [float()],
                InputLength     ::      non_neg_integer(),
                Result          ::      [float()],
                Why             ::      atom().
```

Figure 3.6: Reduce skeleton NIFs

3.1.2 The Skel OCL shared library: `skel_ocl.so`

The native implementation in C++ of the Skel OCL functions is contained in a compiled shared library as requested by Erlang's NIF mechanism (see 2.2.3). Erlang's VM will load at runtime the library and will delegate to it the execution of NIFs.
Both NIF and OpenCL APIs are actually in C, but being C and C++ compatibles, there isn't any problem in using C++.

The source code file for the Skel OCL is `skel_ocl.cpp`. It includes the code to initialize OpenCL, the implementation of the NIFs and the code for handling the OpenCL objects so to expose them as Resource Objects to Erlang.

3.1.3 OpenCL skeletons

Ultimately, the skeletons are implemented as OpenCL kernel, written in OpenCL. A program is the entity that contains a set o kernel functions. In order to be executed, a program must first be compiled. But, since user-specified functions are just kernels, the code from a program must generated.
When a user requests the creation of a certain skeleton, Skel OCL generates the source code for the program appending the user defined kernel to the one implementing the skeleton at hand. The generated code is finally compiled using the compiler provided by OpenCL.
Figure 3.7 shows the program code generated for a map skeleton with a user-defined function that computes the square of the input.

```
#pragma OPENCL EXTENSION cl_khr_fp64: enable

double sq(double x) {
        return x*x;
}

__kernel void MapKernel(__global double* input, __global double * output, unsigned int outputOffset,
unsigned int uiNumElements)
{       __private size_t i = get_global_id(0);
        if(i >= uiNumElements) return;
        output[i + outputOffset] = sq(input[i]);
}
```

Figure 3.7: Generated code for a map skeleton and the user-defined function `sq`

3.2 Example of use

Having introduced Skel OCL and its API, let's now have a look at an example.

In figure 3.8 we use the `mapLL` skeleton and a `sq` kernel to compute the square of each element of a list. We then sum them up thanks to `reduceLL` skeletons and sum kernel.

As we can see, using skeleton is quite easy.

Before calling the actual skeletons we need to create the kernel objects representing our user-defined functions (`sq` and `sum` in this case). Kernel objects for a skeleton are created using the `createKernel` function related to that skeleton: (e.g. `createMapKernel` for map and `createReduceKernel` for reduce).

In this example we are using the LL version of the skeletons; that means that each skeleton, to carry out the required computation will:
un-marshal the input list, copy the data on the device, execute the kernel, copy data back from the device and finally marshal the output list.

Knowing this it's clear that, in the cases where we need to use a skeleton's output as input for another one (i.e. composing them), the whole process is extremely inefficient: not only we have to copy intermediate data from and to the device, but also marshal and un-marshal the same data.

Both of these tasks are really expensive, so we must avoid them whenever is possible.

We will show how to do it in the last section, where we use the lower-level functions provided by Skel OCL to efficiently compose skeletons.

```
example(NumVal) ->

        case isPow2(NumVal) of
                false -> erlang:error(input_not_pow2);
                true -> ok
        end,

        MapFunSrcFile = "sq.cl", MapFunName = "sq",
        ReduceFunSrcFile = "sum.cl", ReduceFunName = "sum",

        PWD = element(2,file:get_cwd()) ++ "/",

        MapFunSrcFileAbsPath =   PWD ++ MapFunSrcFile,
        ReduceFunSrcFileAbsPath = PWD ++ ReduceFunSrcFile,

        InputList = [ X+0.0 || X <- lists:seq(0, NumVal-1) ], %% [0, ..., NumVal-1]

        %%create Kernels objects
        MapSqKernel = checkResult(createMapKernel(MapFunSrcFileAbsPath, MapFunName)),
        ReduceSumKernel = checkResult(createReduceKernel(ReduceFunSrcFileAbsPath, ReduceFunName)),

        %%Map Skeleton: list -> list
        MapOutputList = checkResult(mapLL(MapSqKernel, InputList, NumVal)),

        %%Reduce skeleton: list -> list
        checkResult(reduceLL(ReduceSumKernel, MapOutputList, NumVal))
.
```

Figure 3.8: Example of Skel OCL usage, mapLL and reduceLL skeletons are used

3.3 Implementation details

In this section, we present some interesting details of Skel OCL's implementation.

Firstly we discuss the choice of using Erlang lists as input data for the skeletons.

Secondly we analyze the implementation of the `mapLL skeleton` because it's interesting as an example of skeleton implementation, and for the techniques adopted

to overlap the handling of Erlang lists with the actual computation.

Finally we examine some of OpenCL's and NIF's shortcomings discovered in developing Skel OCL.

3.3.1 Lists vs. Binaries vs. ROs

One of the firsts design decision was choosing which kind of Erlang data structure to support in our skeletons.

Since one of the goals for the library is to be as friendly as possible to Erlang programmers, it was inevitable to choose Erlang lists as the data structure for input and output data.
Lists are, in fact, the cornerstones of functional programming, and Erlang makes no exception; so overlooking lists is unreasonable.

Keeping in mind that to work with OpenCL we need C arrays, it is inevitable to convert the input list to C array and back.
Let *marshaling* be the process of converting an Erlang list to a C-array, and *unmarshaling* the converse.

So we considered and evaluated several ways to marshal and un-marshal a list.

The most straightforward one is using the NIF list handling functions.
Iterating over the list using the `enif_get_list_cell` function we can also, at each step, extract the element inside each list cell. Once we obtained the term representing the element, we can get the content using the `enif_get_double function` (there are `enif_get_ functions` for every Erlang type).
Conversely, to generate a list from a C-array, we firstly create the terms for each element using `enif_make_double`, secondly we pass them as array to `enif_make_list_from_array`, which will generate the list.
See figure 3.9 for an implementation of this technique.

The other technique involves Erlang *binaries*.
A *binary* is a data structure designed for storing large quantities of raw data in a space-efficient manner. They are manipulated using built-in functions (BIFs) or with functions defined in the binary module.
Using binary comprehension it's possible to convert lists of floats into binaries and back, in just a line of code (see figure 3.10).

Binaries are fully supported by the NIF API, and are also implemented as C-arrays so we could use them directly in the native code without any change of representation whatsoever; moreover, converting them to and from lists, is extremely easy.

The last options we examined, was using the Resource Object mechanism (see 2.2.3.1 for an intro to ROs) that we already exploited for exposing OpenCL objects to Erlang. This possibility was ruled out almost immediately owing to ROs totally opaque nature.
ROs are just handles to native data, so they can only be passed back as argument to a NIF. Keeping that in mind, to use the data in Erlang we still need to convert them from and to lists using code similar to the one showed in figure 3.9.

So, to decide between Lists and binaries we set up some tests to measure the performance of the two alternatives.
We measured the time to un-marshal input lists of increasing size and marshaling them back using the two approaches.
The results of the tests (Table in figure 3.11), run on the machine described in section 4.1, show that the technique involving binaries is an order of magnitude slower than the other one.

Moreover, with the list approach, we can work on the list in segments so to overlap the marshaling/un-marshaling with the actual computation on the GPU. In this way, since the costs of converting the list is almost completely hidden by the computation, the conversion process comes essentially for free.

```
ERL_NIF_TERM double_array_to_list(ErlNifEnv *env, double* array, size_t array_size) {

        if(array == NULL)
                return ATOM(error);

        ERL_NIF_TERM* floatTermArray =
                        (ERL_NIF_TERM*) enif_alloc(sizeof(ERL_NIF_TERM) * array_size);

        for(uint i = 0; i < array_size; i++)
                floatTermArray[i] = enif_make_double(env, array[i]);

        ERL_NIF_TERM res = enif_make_list_from_array(env, floatTermArray, array_size);

        enif_free(floatTermArray);

        return res;
}


double* double_list_to_array(ErlNifEnv *env, ERL_NIF_TERM list) {
        double *x;
        if(enif_is_list(env, list)) {

                unsigned int len = 0;
                enif_get_list_length(env, list, &len);

                x = (double*) enif_alloc(len*sizeof(double));

                ERL_NIF_TERM curr_list = list;
                for(int i=0; i<len; i++) {

                        ERL_NIF_TERM hd, tl;
                        enif_get_list_cell(env, curr_list, &hd, &tl);
                        if(!(enif_get_double(env, hd, &(x[i]) ))) {
                                cerr << "ERROR: trying to read float from something else!" << endl;
                        }
                        curr_list = tl;
                }
        } else {
                cerr << "ERROR: input is not list"<< endl;
        }
        return x;
}
```

Figure 3.9: Marshalling/Unmarshalling using NIF API for lists

```
list2bin (L) ->
        << <<X/native-float>> || X <- L >>.

bin2list (B) ->
        [ X || <<X/native-float>> <= B ].
```

Figure 3.10: Converting binaries to lists and vice-versa using binary and list comprehension.

10^	List To List	List To Binary To List
2	3	12
3	38	130
4	575	1200
5	7056	13102
6	71958	245976

Figure 3.11: Running times in microseconds.

We used this streaming technique extensively in implementing map skeletons that works directly on lists, as we are going to see in the next section.

3.3.2 Pipelined Map skeleton: `mapLL`

We now give some details about the implementation of the `mapLL skeleton`. In doing so we will gain some insight into the problems we had to solve during the development of the library.

Following the naming pattern we explained when we earlier introduced skeleton NIFs, `mapLL` is a map skeleton accepting lists as input and returning the result also as lists. This is then the case where marshaling and un-marshaling can potentially take the lion's share of the total running time of a skeleton.

In fact, in order to compute the `mapLL` skeleton the following steps must be completed:

1. *Input un-marshaling* – The elements contained into the input list are extracted and copied into a host buffer

2. *Loading data on the device* – The data is copied from the host buffer to a device buffer on the GPU.

3. *Computation* – The kernel is executed having as input the previous device buffer. The kernel outputs the results in a device buffer.

4. *Unloading data from the device* – The output data is copied from the device back in a host buffer.

5. *Output marshaling* – The output list is created from the data contained into the previous host buffer.

Looking at the figures in Table in figura 3.11 we can immediately understand that, to optimize skeletons' performance, we need to find a way of neutralize (un-)marshaling costs.

To further decrease the running time we exploited the DMA (direct memory access) engine of the GPU. Using DMA is possible to copy data to (or from, but not at the same time, we'd need 2 copy engine for that) while the GPU is busy computing the kernels.

We now recall that, since in a map skeleton there are no data dependencies among the input elements, we can split the input list in an arbitrary number of segments and start computing the user-function on the elements of a segment, independently from the ones belonging to other segments.

Knowing that, we can finally recognize un-marshaling and loading steps (and their inverse unloading and marshaling), as independent from computation, provided that we work on the input in segments.

At this point, it comes almost natural to arrange the three phases we just mentioned, as a three-stage pipeline.

In this way we can overlap computation with both (un-)marshaling and the moving of data to and from the GPU.

As described in 2.1.2, to submit commands for execution on an OpenCL device, command-queues are used. Using in-order queues, to send commands to the device concurrently and independently, more than one command-queue must be associated to a single OpenCL context.

To implement the scheme we just presented, we adopted a dual queue approach.

Each segment of the input is split in two: the first half is processed using the first

queue, and the second using the other one. In doing so we can, for instance, issue a copy command for the first half while executing the kernel on the second one and so on.

A representation of the dual-queue pipelining technique is given in Figure 3.12. For simplicity's sake, in the diagram load is comprised of un-marshaling and copying data from host to device; unload is the opposite process, covering copying from device to host and marshaling the output.

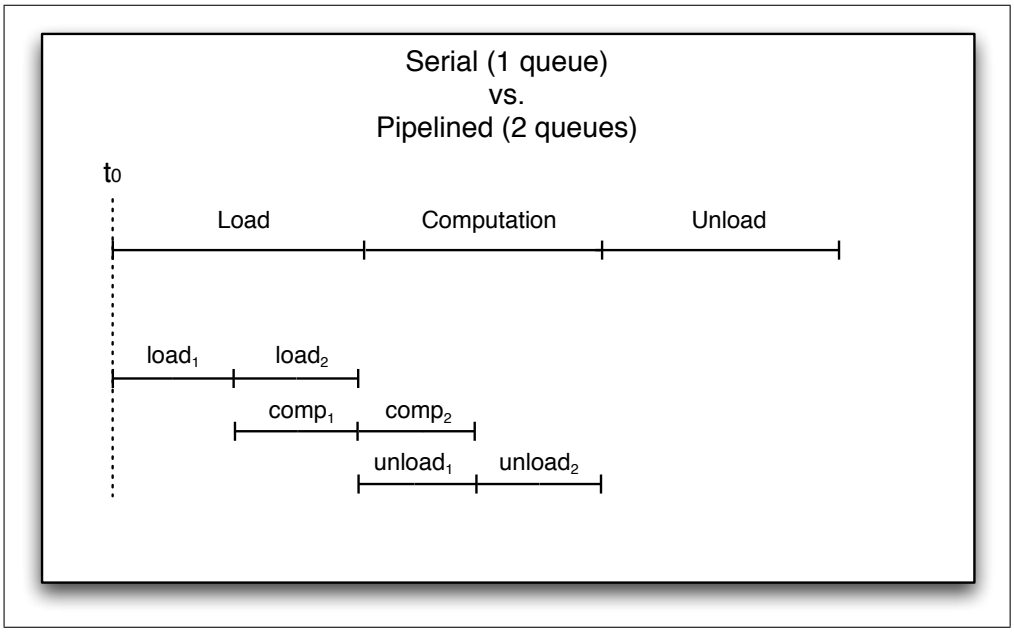

Figure 3.12: Serial vs. pipelined execution using two command queues.

Each of the stage of the pipeline is implemented using the threading facilities provided by the NIF API [2].
Synchronization among the threads is realized exploiting the OpenCL events mechanism: a series of user events (events created calling `clCreateUserEvent`) are used as checkpoints on which a thread must wait until the one responsible for the event fires it (setting them to CL_COMPLETE).

Let's now have a look at the code for the `mapLL` NIF in figure 3.13. It uses the same code structure of every NIF in the library; as mandated by the NIF mecha-

nism, the signature of the function is:

```
static ERL_NIF_TERM mapLL(ErlNifEnv*env,int argc, const ERL_NIF_TERM argv[]);
```

A NIF must be a static function returning an Erlang Term and taking 3 parameters:

1. *env* – A pointer to a process bound environment, all function arguments passed to a NIF and its return value belongs to this environment; it's only valid in the thread where it was supplied as argument until the NIF returns.
2. *argc* – The count of terms available in the argv array.
3. *argv* – An array of Erlang terms representing the actual parameters of the NIF.

After some preliminary checks, we read, one by one, the content of the terms in argv using the NIF API.

For example, `mapLL` first parameter is the kernel ROs (created by calling the `createMapKernel` NIF) denoting the user-function to be executed; therefore we need to get back the pointer to the struct that holds the kernel OpenCL object and we do that calling `enif_get_resource`.

Something similar happens to the remaining NIF's parameters: a list and an integer representing the length of that list.

Since we finally got all the information needed to run the map skeleton, the NIF delegates the execution the `mapLL_Impl` function.

3.3.3 NIF and OpenCL issues

In this section we explain some of the issues we had to solve in implementing the library. In particular we discuss some of OpenCL's and NIF's shortcomings we stumbled upon, and how we overcame them.

NIFs

As we have seen in section 2.2.3, one of the main concerns in developing NIFs is implementing them so that they returns quickly (less than 1 ms) otherwise they would interfere with VM's schedulers blocking other running processes; this is the "dirty NIF" problem.

```
static ERL_NIF_TERM mapLL(ErlNifEnv * env, int argc, const ERL_NIF_TERM argv[]) {

        //mapLL(Kernel::kernel(), InputList::[double()], InputLength::non_neg_integer())

        OCL_INIT_CHECK()
        const uint NUM_ARGS = 3;

        NIF_ARITY_CHECK(NUM_ARGS)
        /*get the parameters (Kernel::kernel(), InputList::[double()], InputLength::non_neg_integer()****/

        kernel_sync* pKernel_s = NULL;
        if (!enif_get_resource(env, argv[0], kernel_sync_rt, (void**) &pKernel_s)) {
                cerr << "ERROR :: mapLL: 1st parameter is not a kernel_sync" << endl ;
                return enif_make_badarg(env);
        }

        ERL_NIF_TERM inputList = argv[1];
        if (!enif_is_list(env, argv[1])) {
                cerr << "ERROR :: mapLL: 2nd parameter is not a list" << endl ;
                return enif_make_badarg(env);
        }

        uint inputListLength;
        if (!enif_get_uint(env, argv[2], &inputListLength)) {

                cerr << "ERROR :: mapLL: 3nd parameter is not a non_neg_integer()" << endl ;
                return enif_make_badarg(env);
        }

        /*****************************************************************************/

        uint pInputC = 1;
        ERL_NIF_TERM* pInputV[1] = { &inputList };

        return mapLL_Impl(env, pKernel_s, (void**) pInputV, pInputC, NULL, inputListLength);

}
```

Figure 3.13: `mapLL` NIF.

This is a limitation to the user of the NIF mechanism, so in release R17B, dirty scheduler have been introduced (see 2.2.3). We chose not to use them being a declaredly experimental feature and also because they require to build the Erlang platform with explicit support for them.

Dirty schedulers aside, as a workaround the NIF documentation suggests dispatching the work to another thread, return from the native function, and wait for the result. Then the thread can send the result back to the calling thread using message passing. We investigated the technique but ultimately we found it unfeasible when working with long lists as with `mapLL`.

As explained in 2.2.3, in the NIF API an environment is a container of terms; a term is valid as long as the environment is valid. There are two types of environments: *process bound and process independent.*

A process bound environment is passed as the first argument to all NIFs; it contains the function parameters and also the term that the NIF must return. Such kind of environment is only valid in the thread where it was supplied as argument until the NIF returns.

Instead, to store terms between NIF calls and to send them to an Erlang process, a process independent environment must be used. Terms can be copied between environments `with enif_make_copy`, and citing the documentation [2]:

All elements of a list/tuple must belong to the same environment as the list/tuple itself.

This last requirement makes the usage of the suggested workaround, prohibitively expensive. In fact, copying between two environments very long lists including each and every one of their elements is way too costly in a high-performance setting.

So, in the end, we decided to ignore the problem taking also into account Skel OCL's intended use, namely as a batch system hence multitasking should not be a concern.

OpenCL

With OpenCL, to get a direct pointer into memory buffer, the `clEnqueueMapBuffer` is used; when done working on the buffer, it must be also unmapped calling `clEnqueueUnmapBufferObject`.

We used this function to access the host buffer containing the input and output data for a computation. In our experiments we found that, on our test machine (section 4.1), (un)mapping OpenCL buffers is a serial activity even using different threads each one working on independent command queues. This means, for example, that it is not possible to (un)map a buffer while computing a kernel, not even when the buffer is on the host and it's unrelated to the kernel: they are executed one after the other. This limitation, forced us to fastidiously design the interactions among `mapLL's` threads so to carry out as many parallel activities as possible. For instance, still in `mapLL`, we overlap the un-mapping of the host buffer used to store input and output data, with the output list creation (`enif_create_list`).

3.4 Example: The `mapReduceLL` Skeleton

We now show an example of efficient skeleton composition; we are going to define a new skeleton, `mapReduceLL`. While presenting it we will highlight some of `Skel OCL`'s features.

`mapReduceLL` is the combination of a map skeleton with a reduce one (see Figure 3.14); taking a list as input, it applies `MapKernel`, then the result is reduced using `ReduceKernel` and finally it returns the reduced value as a one element list.

In this example we compose `mapLD with reduceDL; mapLD` takes a list as input and store the result into a specified device buffer, `reduceDL` instead works on a device buffer and outputs a list.

```
%%------- Skeleton Composition example -----------------

-spec  mapReduceLL(MapKernel, ReduceKernel, {InputList, InputListLength}) -> {ok, Result} when
          MapKernel       ::        kernel(),
          ReduceKernel    ::        kernel(),
          InputList       ::        [float()],
          InputListLength ::        non_neg_integer(),
          Result          ::        [float()]
.

mapReduceLL(MapKernel, ReduceKernel, {InputList, InputListLength}) ->

          NumVal = InputListLength,

          SzDouble = 8,
          SzBufferByte = NumVal * SzDouble,

          MapOutputD = checkResult( allocDeviceBuffer(SzBufferByte, read_write) ),

          checkResult( mapLD(MapKernel, InputList, MapOutputD, InputListLength) ),

          Result = reduceDL(ReduceKernel, MapOutputD),

          releaseBuffer(MapOutputD),

          Result
.
```

Figure 3.14: Skeleton composition example: `mapReduceLL`

Knowing that, to combine the two skeletons, we just need to allocate a suitably sized device buffer, specifying it both as `mapLD`'s output buffer and as the `reduceDL`'s input one.

Along these lines, we can go on composing skeletons at will, having just the care of connecting them using intermediate device buffers.

It is now clear why, the distinction between device and host buffers is meaningful: having the power of managing buffers directly on the device we are able to avoid moving intermediate data back and forth from the GPU, as we naively do in the section 3.2.

Chapter 4

Benchmarking Skel OCL

This chapter presents some applications we developed to test and validate Skel OCL performances. These applications are also useful as an example of how to use the skeletons provided by the library to structure parallel computations.
Every sample application is implemented in three different ways:

1. *Erlang/Skel OCL* – Using the library's facilities to exploit the GPU.
2. *Pure Erlang* – Using Erlang functions from the `lists` module. We compare it against the first one to assess the benefits of using Skel OCL
3. *C++/OpenCL* – This is the reference implementation. Its performance is the target Skel OCL should aim for.

4.1 Test platform and methodology

Let's have a look at the test platform and methodology used to carry out the experiments.
The machine at hand is `pianosa.di.unipi.it` including:

- *CPU* - 8 core/2-way hyperthreading Intel Xeon E5-2650 running at up to 2.80GHz (Refer to [5] for the complete datasheet).
- *RAM* – 32 GB
- *GPU* – NVIDIA GeForce GTX 660 with 2 GB of on-board memory. It's based on NVIDIA's Kepler architecture [21] and is equipped with 5 Streaming Pro-

cessors having 192 CUDA cores each. It is also equipped with one copy engine (see 2.1.6 and [20]).

- *Software* – NVIDIA GeForce driver 304.51 and CUDA 5.5 with OpenCL 1.1 support.

To evaluate Skel OCL performances, we compared the completion time of each implementation of the test applications.

For the ones in Erlang we used the purposely created `test_avg` function; it executes NumRounds times a specified function, timing these calls using Erlang's `timer:tc`. In the C++/OpenCL implementation we used instead the `clock_gettime` function defined in `time.h` getting the wall clock time before and after the calling of the function to measure.

Having the need to simulate various arithmetic intensities in our test applications, we added to user-functions a delay loop that computes `math:sin` a specified number of times. In this way we can easily vary the computation grain and therefore highlight the effect of (un-)marshaling on the total cost of the computation.

Finally, using the running times of the three implementations, we compute the speedup as a measure of the performance improvement achieved using Skel OCL.

For instance, to measure the speedup of Skel OCL versus pure Erlang we use the following formula:

$$speedup = \frac{T_{erl}}{T_{skel\_OCL}}$$

To compare Skel OCL against C++/OpenCL, which we chose as the performance reference, we use:

$$speedup = \frac{T_{Cpp/OCL}}{T_{skel\_OCL}}$$

Since we chose the C++/OpenCL implementation as baseline, its running time is always shorter than Skel OCL's resulting in speedups smaller than 1.

4.2 mapLL

The first application is based on the `mapLL` skeleton; it's an example of the library usage and in testing it we can validate the (un-)marshaling techniques discussed in the implementation section (3.2).

4.2.1 Pure Erlang implementation

In Figure 4.1 we see a pure Erlang implementation, it uses the `lists:map` function to apply to each element of the input list, a user-defined function. The user-defined function we chose, simply computes the square of its argument; as explained in 4.1, there is also a delay loop. The list input contains floats from 0 to `NumVal-1` and it's generated by `lists:seq`.

Finally, we measure the running time of each of the 10 runs, using our test facility for Erlang functions, `test_avg`.

```
-define(ERL_DELAY, 10).

sq(X) ->
        dummyLoadLoop(2.0, ?ERL_DELAY),
        X*X
.

erl_mapLL(NumVal) ->

        io:format("~n-------erl_mapLL TEST--------~n"),
        io:format("Number of values: ~w, Bytes: ~w, Input: seq(0, ~w).~n", [NumVal,NumVal*8,NumVal-1]),
        io:format("Map function: fun sq/1 ~n"),
        io:format("Delay: ~w~n~n", [?ERL_DELAY]),

        L1 = [ X+0.0 || X <- lists:seq(0, NumVal-1) ],

        NumRuns = 10,
        Median = test_avg:test_avg(lists, map, [fun sq/1, L1], NumRuns),

        io:format("erl_mapLL_test median total time over ~w  executions: ~w usec~n", [NumRuns, Median])
.
```

Figure 4.1: `mapLL` skeleton test in pure Erlang implementation

4.2.2 Erlang / Skel OCL

In Figure 4.2 we see the how to implement the same application described in the previous section, using Skel OCL.

The user-defined function is written in OpenCL C; also in this case there's a delay loop to simulate more complex kernels.

The Erlang code is extremely similar; besides creating the kernel (using `createMapKernel`) we just need to replace `lists:map` with `skel_ocl:mapLL`.

```
//OpenCL C kernel in file sq.cl
double sq(double x) {

        return x*x;
}

%%-----Erlang + Skel OCL-------
ocl_mapLL(NumVal) ->

        case isPow2(NumVal) of
                false -> erlang:error(input_not_pow2);
                true -> ok
        end,

        PWD = element(2,file:get_cwd()) ++ "/",

        FunSrcFile = "sq.cl", FunName = "sq",

        FunSrcFileAbsPath = PWD ++ FunSrcFile,

        io:format("~n-------ocl_mapLL TEST--------~n"),
        io:format("Number of values: ~w, Bytes: ~w, Input: seq(0, ~w).~n", [NumVal, NumVal*8, NumVal-1]),
        io:format("Map function file: ~p~n", [FunSrcFileAbsPath]),
        io:format("Map function name: ~p~n~n", [FunName]),

        L1 = [ X+0.0 || X <- lists:seq(0, NumVal-1) ],

        MapSqKernel =
                checkResult( createMapKernel(FunSrcFileAbsPath, FunName)),

        NumRounds = 10,

        Median = test_avg:test_avg(skel_ocl, mapLL, [MapSqKernel, L1, NumVal], NumRounds),

        io:format("ocl_mapLL_test median total time over ~w  executions: ~w usec~n", [NumRounds, Median])
.
```

Figure 4.2: `mapLL` skeleton test in Erlang / Skel OCL

4.2.3 C++/OpenCL

In figure 4.3 we see a C++/OpenCL implementation for the map skeleton. We use it to find a baseline for comparing the performance of the other two implementations.

The map skeleton is implemented in the `map_impl` function, which is a refactored version of the one in Skel OCL.

4.2.4 Tests Results

We tested mapLL on input lists of one million (2^{10}) elements, we run the test 10 times for 0, 20, 40 and 80 delay loops so to simulate increasingly complex kernels.

Figure 4.4 shows the speedup of Skel OCL (4.2.2) versus the pure Erlang (4.2.1) implementation of `mapLL` skeleton, while Figure 4.2.4.2 shows the one of Skel OCL relative to the C++/OpenCL version.

As evident from the graphs, to achieve significant speedups the kernel to compute must be complex enough to cover the cost of list (un-)marshaling.

Thanks to the techniques described in 3.2.2 to overlap (un-)marshaling to kernel execution, we neutralize the cost of the former when the cost of the latter is greater.

Turning our attention at figure 4.5, where we see the speedup curve relative to the C++/OpenCL reference implementation, we notice that we get to the breakeven point around the value of 40 delay loops.

As we increase the number of delay loops, the curve flattens after the value 40, showing that the speedup is approaching its limit. That is, what we lose to the reference implementation is due just to Erlang's intrinsic inefficiencies, and not to (un-)marshaling cost.

```
int main(int argc, char** argv) {

	const uint NUM_ELEMS_EXP = 10;

	if(argc == 2) NUM_ELEMS_EXP = (uint) atoi(argv[1]);
	else {
		cout << "\nUsage: map_test NUM_ELEMS_POW2_EXPONENT\n";
		cout << "Using default value: 2^10\n";
	}

	if(!init_OCL()) { cout<<"Can't initialize OpenCL"; return 0; }

	const uint TEST_RUNS = 10;

	const string mapFunFile = "sq.cl", mapFunName = "sq";

	const uint NUM_ELEMS = pow2(NUM_ELEMS_EXP);

	cout << "\n-------ocl_map TEST-------\n";
	cout << "Number of values: " << NUM_ELEMS << ", Bytes: "<< NUM_ELEMS * 8 <<", Input: seq(0, "
	<< NUM_ELEMS-1 <<").\n";
	cout << "Map function file: "<< mapFunFile << "\n";
	cout << "Map function name: "<< mapFunName <<"\n\n";
	cout<< TEST_RUNS <<" rounds test\nRuntimes in usecs\n\n";

	//init buffer with double values from 1 to arrayLen
	const double* input = seqArray(NUM_ELEMS);

	timespec runs_counters[TEST_RUNS+1];

	for (uint i = 0; i < TEST_RUNS; ++i) {
		GET_TIME(runs_counters[i]);

		map_impl(input, NUM_ELEMS, mapFunFile, mapFunName);
	}
	GET_TIME(runs_counters[TEST_RUNS]);

	delete[] input;

	long runs_times[TEST_RUNS];

	long total = 0;

	cout <<"[ ";
	for (int i = 0; i < TEST_RUNS; ++i) {
		total += runs_times[i] = DIFF(runs_counters[i],runs_counters[i+1]);
		cout << runs_times[i] << " ";
	}
	cout <<"]";

	sort(std::begin(runs_times), std::end(runs_times));

	cout << "\nMedian: " << runs_times[(int)round(TEST_RUNS/2)]<< endl;
	cout << "Avg: " << total/TEST_RUNS<< endl<<endl;

	return 0;
}
```

Figure 4.3: Map skeleton test C++/OpenCL implementation

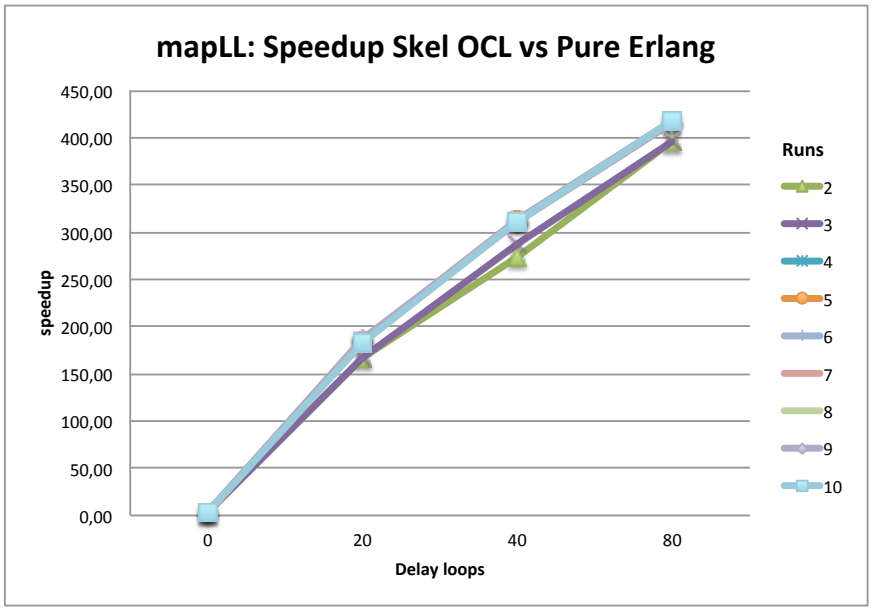

Figure 4.4: Speedup of Skel OCL vs. Pure Erlang implementation of `mapLL` skeleton

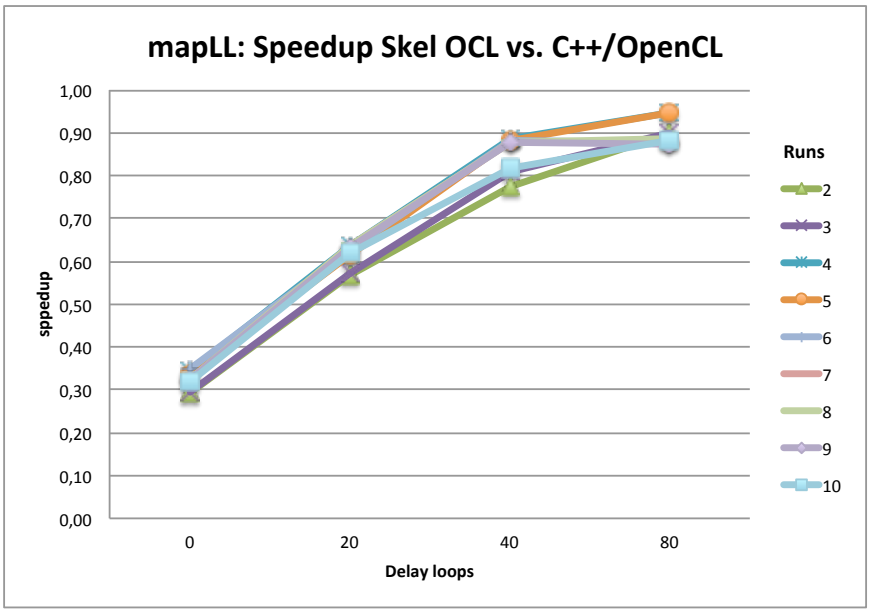

Figure 4.5: Speedup of Skel OCL vs. C++/OpenCL Erlang implementation of `mapLL` skeleton

4.3 Numerical Integration

The second application is numerical integration using the midpoint method:

$$\int_a^b f(x)\,dx \approx h\sum_{j=1}^{n} f(m_j)$$

$$where\ \ h = \frac{(a-b)}{n}\ \ and\ \ m_j = a + (j - \frac{1}{2})\ \ h, 1 \leq j \leq m$$

4.3.1 Pure Erlang implementation

We implement the numerical integration application using `lists:map` and `lists:foldl` combining them in an Erlang implemented `mapReduce` skeleton similar to the one presented in 3.3.

Firstly we generate a list containing the m_j points, secondly we map the integrand function over it and finally we multiply the sum of the function values by h, obtaining the approximation of the integral.

The code for this implementation is shown in figure 4.6.

4.3.2 Erlang / Skel OCL

In figure 4.7 we see the implementation in Erlang / Skel OCL. Here we exploit the `mapReduceLL` skeleton presented in section 3.3 to compute the numerical integral in the same way we described in the previous section.

Like in the `mapLL` example, kernels creation aside, the code is extremely similar to the one in pure Erlang.

4.3.3 C++/OpenCL

In figure 4.8 we see a C++/OpenCL implementation for the numerical integration using skeletons as described earlier. As before, we use it to find a baseline for comparing the performance of the other two implementations.

```
%%Pure erlang implementation
integrandFun(X) ->
        dummyLoadLoop(2.0,?ERL_DELAY),
        X+2
.

erl_integr_test(A,B,N_nodes) ->

        Integrand = fun integrandFun/1,
        Nodes_list = generate_nodes(A, B, N_nodes),

        io:format("~n-------erl_integr_test--------~n"),
        io:format("Integrand fun: \"fun(X)-> X+2 end\"~n"),
        io:format("Delay: ~w~n", [?ERL_DELAY]),
        io:format("Bounds: [~w,~w]~n",[A,B]),
        io:format("Number of Nodes: ~w~n~n",[N_nodes]),

        Runs = 10,

        Median = test_avg:test_avg(integration, integrate, [erl, Integrand, A, B, {Nodes_list, N_nodes}], Runs),
        io:format("erl_integr_test median total time (~w rounds): ~w usec~n", [Runs, Median])
.
generate_nodes(A, B, N_nodes) ->

        H = (B-A)/N_nodes,

        MidPoint = fun(I) -> A + ( (I+1.0) - 0.5 ) * H end,

        Points_method = MidPoint,

        for(0, N_nodes-1, Points_method)  %% from 0 to n-1

.

%% used also to test Skel OCL
integrate(Type, Integrand, A, B, {Nodes_list, N_nodes}) ->

        H = (B-A)/N_nodes,

        Sum =
                case Type of
                        erl -> erl_integr_mapReduce(Integrand, {Nodes_list, N_nodes});
                        ocl -> ocl_integr_mapReduce(Integrand, {Nodes_list, N_nodes});
                        _ -> erlang:error(wrong_type_error)
                end,

        Sum*H
.

erl_integr_mapReduce(F, {Nodes_List, _N_nodes}) ->

        Sum_fun = fun(X,Y)-> X+Y end,

        Values_list = lists:map(F, Nodes_List),
        Sum = lists:foldl(Sum_fun, 0.0, Values_list),

        Sum
.
```

Figure 4.6: Numerical integration using pure Erlang

```
%%Numerical Integration implemented whith skel_ocl mapReduceLL.
ocl_integr_test(A, B, N_nodes) ->

        Integrand = "f_integrand",%name of the function to be integrated, must be the same of .cl kernel file
        Nodes_list = generate_nodes(A, B, N_nodes),

        io:format("~n-------ocl_integr_test-------~n"),
        io:format("Integrand OCL Kernel file: ~p~n",[Integrand++".cl"]),
        io:format("Bounds: [~w,~w]~n",[A,B]),
        io:format("Numeber of Nodes: ~w~n~n",[N_nodes]),

        Runs = 10,

        Median = test_avg:test_avg(integration, integrate, [ocl, Integrand, A, B, {Nodes_list, N_nodes}], Runs),
        io:format("ocl_integr_test median total time (~w rounds): ~w usec~n", [Runs, Median])
.

ocl_integr_mapReduce(IntegrandFunName, { Nodes_List, N_nodes }) ->

        ReduceFunSrcFile = "sum.cl", ReduceFunName = "sum",

        PWD = element(2,file:get_cwd()) ++ "/",

        %%The integrand's src file has the same name of the integrand function
        Integrand_SrcFileAbsPath =  PWD ++ IntegrandFunName ++ ".cl",

        ReduceFunSrcFileAbsPath = PWD ++ ReduceFunSrcFile,

        IntegrandMapKernel = checkResult(createMapKernel(Integrand_SrcFileAbsPath, IntegrandFunName)),
        ReduceSumKernel = checkResult(createReduceKernel(ReduceFunSrcFileAbsPath, ReduceFunName)),

        mapReduceLL(IntegrandMapKernel, ReduceSumKernel, {Nodes_List, N_nodes}),
.
```

Figure 4.7: Numerical Integration test in Erlang / Skel OCL

The mapReduce skeleton is implemented in the `mapReduce_impl` function, which is the composition of refactored versions of Skel OCL's `mapDD` and `reduceDD` skeletons.

4.3.4 Tests Results

We computed an approximation of

$$\int_{10}^{20} (x+2)\,dx$$

on 1 million points (2^{10}) using the method described in 4.3.1.

We run the test 10 times for 0, 20, 40 and 80 delay loops so to simulate increasingly heavier computational loads.

Figure 4.9 shows the speedup of Skel OCL (4.3.2) versus the pure Erlang (4.3.1) implementation of the numerical integration test application, while Figure 4.10 shows the one of Skel OCL relative to the C++/OpenCL version.

Also here the kernel load is not heavier enough to achieve a significant speedup so we need the delay loop.

In this application we get higher speedups thanks to the highly optimized reduce kernel based on a NVIDIA reference implementation [18].

Again, we see in Figure 4.3.4.2 how marshaling cost stops being the performance bottleneck after 40 delay loops.

4.4 Dot product

The last test application computes the dot product of two n-dimensional vectors u and v using the formula:

$$u * v = u_1v_1 + u_2v_2 + \cdots + u_nv_n = \sum_{j=1}^{n} f(u_jv_j)$$

```
double mapReduce_impl(double** inputs, uint inputC, uint INPUT_LEN, const string& mapFunFile, const
string& mapFunName, const string& reduceFunFile, const string& reduceFunName) ;

int main(int argc, char** argv) {

        uint N_NODES_EXP = 10;

        if(argc == 2)
                N_NODES_EXP = (uint) atoi(argv[1]);
        else {
                cout << "\nUsage: integration_test N_NODES_POW2_EXPONENT\n";
                cout << "Using default value: 2^10\n";
        }

        if(!init_OCL()) {
                cout<<"Can't initialize OpenCL";
                return 0;
        }

        const double A = 10, B = 20;//integration bounds
        const uint N_NODES = pow2(N_NODES_EXP);

        const uint TEST_RUNS = 10;

        const string mapFunFile = "f_integrand.cl", mapFunName = "f_integrand";
        const string reduceFunFile = "sum.cl", reduceFunName = "sum";

        cout << "\n-------ocl_integr TEST-------\n";
        cout << "Integrand fun: "<< mapFunName << "\n";
        cout << "Bounds: ["<< A <<"," << B <<"]\n";
        cout << "Number of Nodes: " << N_NODES << "\n\n";

        cout<< TEST_RUNS <<" rounds test\nRuntimes in usecs\n\n";

        //generate nodes
        const double H = (B-A)/N_NODES;

        double* nodes = new double[N_NODES];

        for (int i = 0; i < N_NODES; ++i)
                nodes[i] = A + ( (i+1.0) - 0.5 ) * H;

        //test runs
        timespec runs_counters[TEST_RUNS+1];

        for (uint i = 0; i < TEST_RUNS; ++i) {
                GET_TIME(runs_counters[i]);

                mapReduce_impl(&nodes, 1, N_NODES, mapFunFile, mapFunName, reduceFunFile,
                reduceFunName);

        }
        GET_TIME(runs_counters[TEST_RUNS]);

        ......
        return 0;
}
```

Figure 4.8: Numerical Integration test in C++/OpenCL

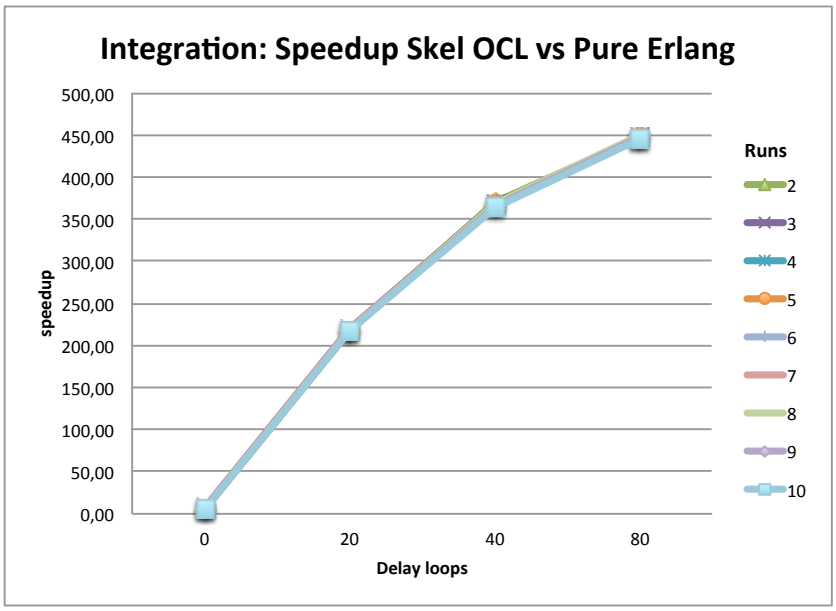

Figure 4.9: Speedup of Skel OCL vs. Pure Erlang implementation of numerical integration

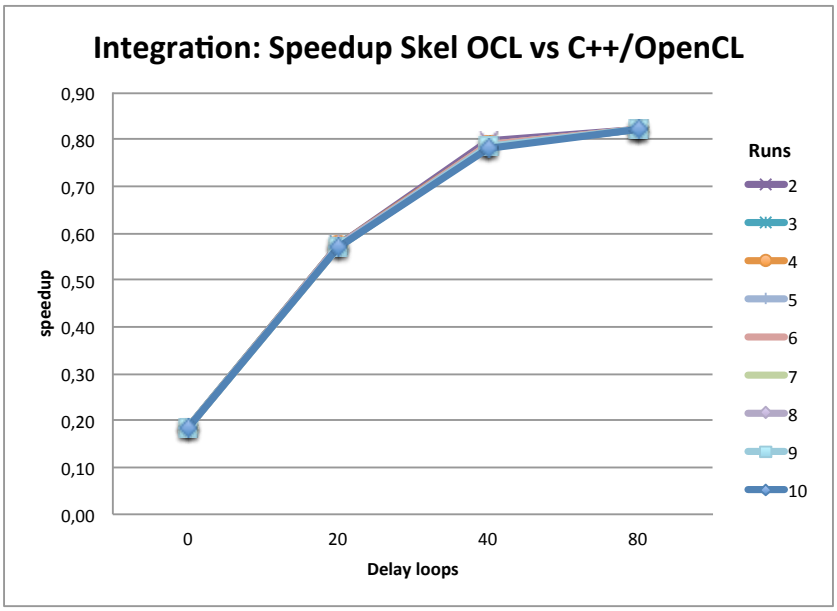

Figure 4.10: Speedup of Skel OCL vs. C++/OpenCL implementation of the numerical integration

We can easily express this computation using the binary map and reduce kernel.

The binary Map skeleton multiplies the corresponding elements of the two vectors and then the Reduce skeleton sums all the intermediate products resulting in the dot product of the two vectors.

4.4.1 Pure Erlang implementation

To implement the computation of the dot product of two vectors represented as lists, we used `lists:zipWith` and `lists:foldl`.

`ZipWith` is the Erlang counterpart of Skel OCL `map2` skeleton; it combines the corresponding elements of two lists of equal length into one list. To each input pair a specified binary function is applied; the result is then appended to the output list.

The code for this implementation is shown in figure 4.11.

4.4.2 Erlang / Skel OCL

Figure 4.12 shows the implementation in Erlang / Skel OCL.

Here we see another instance of skeleton composition: we compose `map2LD` skeleton with a `reduceDL` through a device buffer as we did in section 3.4 where we implemented the `MapReduceLL` skeleton.

The kernel required by a map2 skeleton must be created using the three-argument version of `createMapKernel` specifying the arity of the function, in this case 2.

4.4.3 C++/OpenCL

In figure 4.13 we see a C++/OpenCL implementation for the dot product application using skeletons as described earlier. As always, we use it to find a baseline for comparing the performance of the other two implementations.

The mapReduce skeleton is implemented in the `mapReduce_impl` function, which is the composition of refactored versions of Skel OCL's `mapDD` and `reduceDD` skeletons.

```
%%%----------pure Erlang version----------------

sum(X,Y) -> X+Y.

zipFun(X,Y) ->
        dummyLoadLoop(2.0,?ERL_DELAY),
 X*Y.

erl_dotProduct(V1, V2) ->

        Sums = lists:zipwith(fun zipFun/2, V1, V2),

        lists:foldl(fun sum/2, 0.0, Sums)
.

erl_dotProduct_test(NumVal) ->

        Val = 2,

        io:format("~n-------erl_dotProduct TEST--------~n"),
        io:format("Vectors' dimensions: ~w~nV1:seq(0,~w)~nV2:duplicate(~w,~w): ~n", [NumVal, NumVal-1,
        NumVal, Val]),
        io:format("Delay: ~w~n~n", [?ERL_DELAY]),

        V1 = [ X+0.0 || X <- lists:seq(0,NumVal-1) ],
        V2 = [ X+0.0 || X <- lists:duplicate(NumVal,Val) ],

        Runs = 10,

        Median = test_avg:test_avg(dotProduct, erl_dotProduct, [V1, V2], Runs),
        io:format("erl_dotProduct_test median total time (~w rounds): ~w usec~n", [Runs, Median])
.
```

Figure 4.11: Erlang implementation of the Dot Product test application

```
ocl_dotProduct(NumVal, V1, V2) ->

        SzDouble = 8,
        SzBufferByte = NumVal * SzDouble,

        MapFunSrcFile = "mul.cl", MapFunName = "mul",
        ReduceFunSrcFile = "sum.cl", ReduceFunName = "sum",

        PWD = element(2,file:get_cwd()) ++ "/",

        MapFunSrcFileAbsPath =   PWD ++ MapFunSrcFile,
        ReduceFunSrcFileAbsPath = PWD ++ ReduceFunSrcFile,


        %%Create and compile kernels
        MapMulKernel = checkResult(createMapKernel(MapFunSrcFileAbsPath, MapFunName, 2)),
        ReduceSumKernel = checkResult(createReduceKernel(ReduceFunSrcFileAbsPath, ReduceFunName)),

        %%allocate work buffer on device
        MapOutputD = checkResult(allocDeviceBuffer(SzBufferByte, read_write),

        checkResult( map2LD(MapMulKernel, V1, V2, MapOutputD, NumVal) ),

        Result = checkResult( reduceDL(ReduceSumKernel, MapOutputD) ),

        releaseBuffer(MapOutputD),

        Result
.
ocl_dotProduct_test(NumVal) ->

        %% must be a power of 2
        case isPow2(NumVal) of
                false -> erlang:error(input_not_pow2);
                true -> ok
        end,
        Val = 2,

        io:format("~n-------ocl_dotProduct TEST-------~n"),
        io:format("Vectors' dimension: ~w~nV1 = seq(0,~w)~nV2 = duplicate(~w,~w): ~n~n", [NumVal,
        NumVal-1, NumVal, Val]),

        V1 = [ X+0.0 || X <- lists:seq(0,NumVal-1) ],
        V2 = [ X+0.0 || X <- lists:duplicate(NumVal,Val) ],

        Runs = 10,

        Median = test_avg:test_avg(dotProduct, ocl_dotProduct, [NumVal, V1, V2], Runs),
        io:format("ocl_dotProduct_test median total time (~w rounds): ~w usec~n", [Runs, Median])
.
```

Figure 4.12: Dot Product test application Erlang / Skel OCL implementation

```
double mapReduce_impl(double** inputs, uint inputC, uint INPUT_LEN, const string& mapFunFile, const string&
mapFunName, const string& reduceFunFile, const string& reduceFunName) ;

int main(int argc, char** argv) {

	uint NUM_ELEMS_EXP = 10;

	if(argc == 2)
		NUM_ELEMS_EXP = (uint) atoi(argv[1]);
	else {
		cout << "\nUsage: dotProduct_test DIMENSION_POW2_EXPONENT\n";
		cout << "Using default value: 2^10\n";
	}

	if(!init_OCL()) {
		cout<<"Can't initialize OpenCL";
		return 0;
	}

	const uint NUM_ELEMS = pow2(NUM_ELEMS_EXP);

	const uint TEST_RUNS = 10;

	const string mapFunFile = "mul.cl", mapFunName = "mul";
	const string reduceFunFile = "sum.cl", reduceFunName = "sum";

	cout << "\n-------ocl_dotProduct TEST-------\n";
	cout << "Vectors' dimensions: " << NUM_ELEMS << " V1 = V2 : seq(0, "<< NUM_ELEMS-1 <<")\n";

	cout<< TEST_RUNS <<" rounds test\nRuntimes in usecs\n\n";

	double** inputs = new double*[2];
	inputs[0] = new double[NUM_ELEMS];
	inputs[1] = new double[NUM_ELEMS];

	for (int i = 0; i < NUM_ELEMS; ++i) {
		inputs[0][i] = (double) i;
		inputs[1][i] = 2.0;
	}
	//test runs
	timespec runs_counters[TEST_RUNS+1];

	for (uint i = 0; i < TEST_RUNS; ++i) {
		GET_TIME(runs_counters[i]);

		mapReduce_impl(inputs, 2, NUM_ELEMS, mapFunFile, mapFunName, reduceFunFile,
		reduceFunName);
	}
	GET_TIME(runs_counters[TEST_RUNS]);

	....
	return 0;
}
```

Figure 4.13: Dot Product test application in C++/OpenCL

4.4.4 Tests Results

To test the Dot Product application we computed the dot product of two 2^{10} dimensional vectors u and v:

$$u * v = \begin{pmatrix} 0 \\ 1 \\ \vdots \\ 2^{10} - 1 \end{pmatrix} * \begin{pmatrix} 2 \\ 2 \\ \vdots \\ 2 \end{pmatrix}$$

As usual, we run the test 10 times for 0, 20, 40 and 80 delay loops so to simulate increasingly heavier computational loads.

Figure 4.14 shows the speedup of Skel OCL (4.4.2) versus the pure Erlang (4.4.1) implementation of the dotProduct test application, while Figure 4.15 shows the one of Skel OCL relative to the C++/OpenCL version.

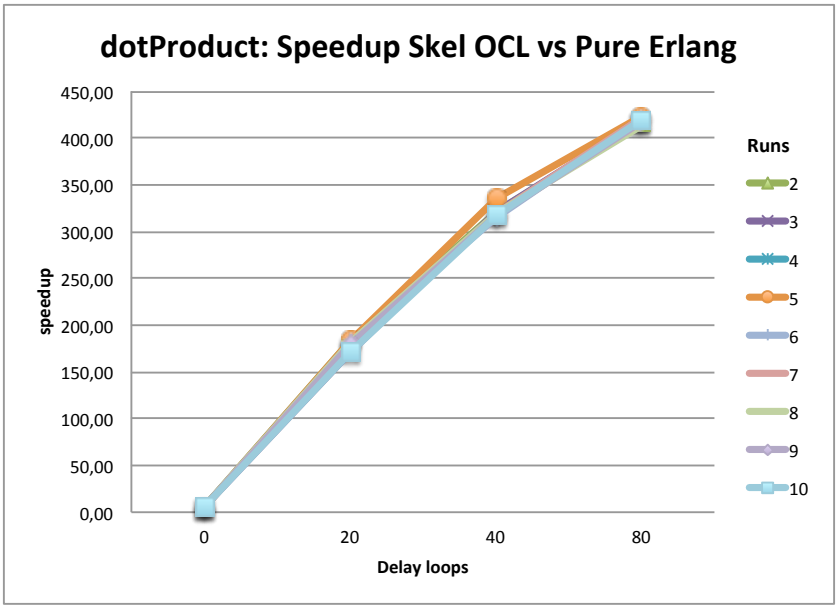

Figure 4.14: Speedup of Skel OCL vs. Pure Erlang implementation of dotProduct

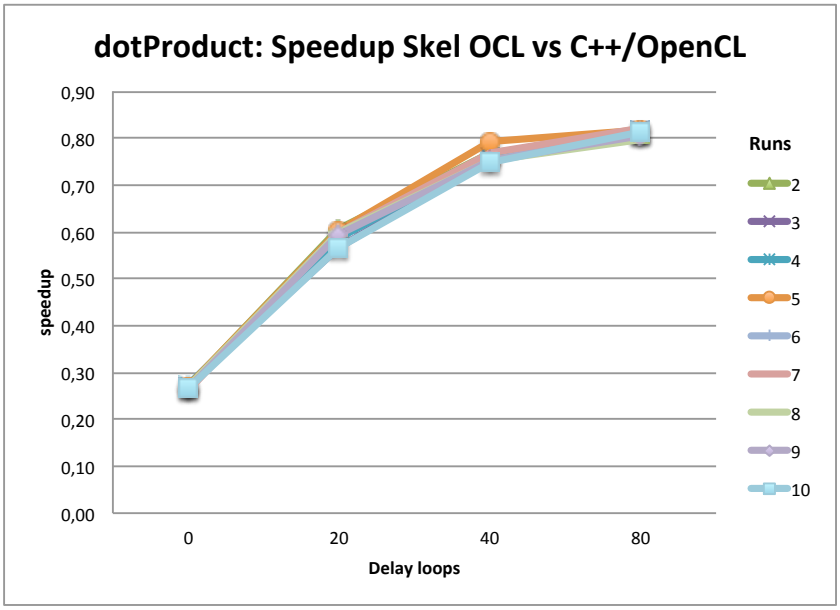

Figure 4.15: Speedup of Skel OCL vs. C++/OpenCL implementation of dotProduct

4.5 Conclusion

In this chapter we presented three applications we developed to test and validate Skel OCL performances:

- *MapLL* – we analyzed the costs of (un-)marshaling, evaluating the techniques discussed in 3.2.
- *Numerical Integration* – An example of usage of the `MapReduceLL` skeleton defined in 3.3.
- *Dot Product* - An example of skeleton composition using Skel OCL low-level function and of the usage of map2 skeleton.

The outcome of the experiments we presented, shows, on one hand, a significant performance improvement over pure Erlang implementations and, on the other hand, an acceptable performance loss over C++/OpenCL reference implementations.

Chapter 5

Conclusion

In this thesis we presented the Skel OCL skeleton library for the Erlang programming language.

The goal of the project we developed in this thesis was to provide Erlang with an easy to use, high performance, data parallel skeleton library exploiting GPUs' processing power. A prime requirement was vendor neutrality, so OpenCL was the only possible choice.
The skeletons should be easy to use, possibly be a nearly perfect drop-in replacement to standard Erlang functions with similar semantic.

In chapter 1 we discussed how the rise of ubiquitous parallel hardware called for new programming methodologies.
In particular we focused our attention on on GPUs, which, from their origins as special purpose devices, evolved into fully-fledged parallel computing engines.

In chapter 2 we introduced the technologies we used in the project.
To work with GPUs, we chose the OpenCL programming framework thanks to its industry standard status and broad support by the vendors.
As the target programming language we chose Erlang: developed in the telecommunication industry, Erlang has seen a rapid increase in adoption thank to its built-in support for concurrency and distribution.

Erlang is also one of the focuses of the European Paraphrase project. The Paraphrase project is particularly interested in Erlang's support for parallel processors including GPUs.

In chapter 3 we presented Skel OCL, discussing its architecture and implementation. We showed that Skel OCL is an Erlang NIF library providing the map and reduce data parallel skeletons to execute user-defined functions on a GPU. In addition to the built-in skeleton Skel OCL also provides low-level functions to efficiently combine them.

Chapter 4 presented some applications we developed to test and validate Skel OCL performances. In addition to a Skel OCL implementation, we also saw a pure Erlang and a C++/OpenCL one; comparing the completion times of the three we evaluated Skel OCL performances.
Analyzing the tests' results we found out that using Skel OCL we achieved a significant performance improvement over pure Erlang, without losing too much to the C++/OpenCL implementation we chose as baseline.

In summary, we developed Skel OCL, a prototype Erlang NIF library using OpenCL. Skel OCL provides data parallel skeletons to accelerate computations on lists of Erlang floats.

The user code for such computations is specified writing OpenCL kernels using the OpenCL C (see 2.1.5) programming language.

The library has a very user-friendly API being it modeled after Erlang's list module, with which every Erlang programmer is familiar.

Skel OCL also provides lower-level functions for efficient skeleton composition, using these functions the user can easily define new skeletons having the same efficiency as the built-in ones (see 3.3 for an example).

We finally validated the design and implementation of Skel OCL carrying out experiments using three test applications.
The outcome of the tests (chapter 4) shows, on one hand, a significant performance improvement over a pure Erlang implementation and, on the other hand, an acceptable performance loss over a C++/OpenCL reference implementation.

Future work

Owing to its prototypal nature, the directions along witch Skel COL can be further developed are numerous, for example:

- *Broader data types support* – increase the number of data types supported by the skeletons. Skel OCL currently supports computations on lists of Erlang floats. A wider choice of data types would improve Skel OCL suitability, increasing the number of scenarios.

- *More skeletons* – Provide additional optimized built-in skeletons either to be used directly or to be combined employing Skel OCL skeletons composition mechanisms.

- *Kernels in Erlang* – Currently, user-defined functions must be written in OpenCL C so the option of implementing kernels directly in Erlang would be desirable to shorten the learning curve.

Bibliography

[1] Erlang lists module documentation. http://www.erlang.org/doc/man/lists.html.

[2] Erlang NIF module documentation. http://www.erlang.org/doc/man/erl_nif.html.

[3] Erlang Programming Language. http://www.erlang.org.

[4] Erlang/OTP Documentation. http://www.erlang.org/doc/.

[5] Intel Xeon E5-2650 specs. http://ark.intel.com/products/64590/Intel-Xeon-Processor-E5-2650-20M-Cache-2_00-GHz-8_00-GTs-Intel-QPI.

[6] OpenCL, the open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/.

[7] Skel CL: A Skeleton Library for Heterogeneous Systems. http://skelcl.uni-muenster.de/.

[8] SkeTO project. http://sketo.ipl-lab.org/.

[9] The FastFlow parallel programming framework. http://calvados.di.unipi.it/dokuwiki/doku.php/ffnamespace:about.

[10] The Paraphrase project. http://www.paraphrase-ict.eu/.

[11] The SkePU Skeleton Programming Framework. http://www.ida.liu.se/~chrke/skepu/.

[12] AMD. AMD OpenCL website. `http://developer.amd.com/tools-and-sdks/opencl-zone`.

[13] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. 2 edition, 2013.

[14] M. Cole. Algorithmic Skeletons: Structured Management of Parallel Computations. volume Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[15] M. Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.

[16] J. Enmyren and C. W. Kessler. SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems. In *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, HLPP '10, pages 5–14, New York, NY, USA, 2010. ACM.

[17] A. Munshi, B. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg. *OpenCL Programming Guide*. 2011.

[18] NVIDIA. NVIDIA CUDA Parallel Reduction sample. `http://docs.nvidia.com/cuda/cuda-samples/index.html#cuda-parallel-reduction`.

[19] NVIDIA. NVIDIA CUDA Toolkit documentation. `http://docs.nvidia.com/cuda/`.

[20] NVIDIA. NVIDIA GeForce GTX-660 specs. `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-660/specifications`.

[21] NVIDIA. NVIDIA Kepler GK110 Architecture. `http://www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-Architecture-Whitepaper.pdf`.

[22] F. Rabhi and S. Gorlatch. *Patterns and skeletons for parallel and distributed computing*. Springer Science & Business Media, 2003.

Appendix A

Usage example

In this appendix we are going to see how to get started with Skel OCL.

We show how to setup the library, define an Erlang function that uses a skeleton and the OpenCL C kernel for the computation.

First of all we need to build Skel OCL shared library using `make`. Several target are available, but here we are just interested in the default one.
Using the default target we force Skel OCL to look, during initialization, for a NVIDIA GPU, and to fail when none are found.
So invoking the make command from the shell we build the shared library skel_ocl.so.

Now we need to write an OpenCL C kernel that implements our computation.
We have to put the functions definitions in a file. In this example we define a simple kernel that performs arithmetic instructions in a file named `userFun.cl` (Figure A.1)

The next step is to write the Erlang code for our computation (Figure A.2). Using the high-level skeletons provided by Skel OCL is really easy.

The skeletons working on Erlang lists (the `LL` type) just need three parameters:

1. The user-defined function as a kernel object
2. The input list (2 in case of `map2`)
3. The length of the input list.

So we firstly to create the kernel object using one of the available NIF. Since we are going to use it in a map skeleton, what we need is `createMapKernel`: As with any `createKernel` NIF we have to pass as arguments the path to the source file and the name of the kernel function.

We now need a floats input list; currently there are limitations on the length of input lists: they must have a number of elements that is a power of two and mustn't be too short (how short depends on the OpenCL device used).

Having all set up, we the mapLL skeleton and obtain the list containing the results of our computation.

Lastly, before executing `exampleFun/1` (e.g. invoking it from the Erlang shell) we need to call `skel_ocl:cl_init()` to initialize the library.

```
//foo kernel function
double foo(double x){

        return (x+1)*2;

}
```

Figure A.1: The `foo` kernel function in `exampleKernel.cl`

```
-module(usageExample).

-export([exampleFun/1]).


-import(erl_utils,
                [
                 isPow2/1,
                 pow2/1
                ]).

-import(skel_ocl,
                [checkResult/1,
                 createMapKernel/2,
                 mapLL/3
                ]).

exampleFun(NumVal) ->

        case isPow2(NumVal) of
                false -> erlang:error(input_not_pow2);
                true -> ok
        end,

        MapFunSrcFile = "exampleKernel.cl", MapFunName = "foo",

        PWD = element(2,file:get_cwd()) ++ "/",

        MapFunSrcFileAbsPath =   PWD ++ MapFunSrcFile,

        InputList = [ X+0.0 || X <- lists:seq(0, NumVal-1) ],

        %%create Kernels objects
        MapSqKernel = checkResult(createMapKernel(MapFunSrcFileAbsPath, MapFunName)),

        %%Map Skeleton: list -> list
        OutputList = checkResult(mapLL(MapSqKernel, InputList, NumVal))
.
```

Figure A.2: The `exampleUsage` module

Appendix B

Skel OCL: Source Code

This appendix contains the Skel OCL source code. Two are the main sections:

B.1 *skel_ocl.erl* - The Erlang part of the library, containing the API definitions.

B.2 *skel_ocl.so* The native implementation of Skel OCL NIFs, including C++ NIFs and OpenCL C kernel code

B.1 skel_ocl.erl

```
1  -module(skel_ocl).
2  -on_load(init/0).
3
4  -export([
5   %%initializations NIFs
6    init/0,
7    cl_init/0,
8    cl_release/0,
9    skeletonlib/1,
10   checkResult/1,
11
12   %%Buffer NIFs
13   listToBuffer/2,
14   bufferToList/1,
15   bufferToList/2,
16   getBufferSize/1,
17   allocDeviceBuffer/2,
18   allocHostBuffer/1,
19   releaseBuffer/1,
20   copyBufferToBuffer/2,
```

```
21    copyBufferToBuffer/3,
22
23    %%Program and Kernel NIFs
24    buildProgramFromString/1,
25    buildProgramFromFile/1,
26    createKernel/2,
27
28    %Map NIF
29    createMapKernel/2,
30    createMapKernel/3,
31    createMapKernel/4,
32
33    mapDD/3,
34    mapLL/3,
35    mapLD/4,
36
37    map2DD/4,
38    map2LD/5,
39    map2LL/4,
40
41    %MapReduce
42    mapReduceLL/3,
43
44    %Reduce NIF
45    createReduceKernel/2,
46    createReduceKernel/3,
47
48    reduceDD/3,
49    reduceDL/2,
50    reduceLL/3
51
52    ]).
53
54 init()->
55    erlang:load_nif("./skel_ocl",0).
56
57
58
59 %%Buffer Types and functions ------------------------------------
60
61 -type hostBuffer()     ::  binary().
62 -type deviceBuffer()   ::  binary().
63 -type buffer()         ::  hostBuffer() | deviceBuffer().
64
65 -type rw_flag()        ::  read | write | read_write.
66
67 %%Get the size of Buffer in Bytes
```

```
68 -spec getBufferSize(Buffer) -> {ok, SizeByte} | {error, Why} when
69     SizeByte  ::  non_neg_integer(),
70     Buffer    ::  buffer(),
71     Why       ::  atom().
72
73 getBufferSize(_) -> {error, ecl_lib_not_loaded}.
74
75
76 %%Allocate a buffer on the host of size SizeByte bytes
77 -spec allocHostBuffer(SizeByte) -> {ok, Buffer} | {error, Why} when
78     SizeByte  ::  non_neg_integer(),
79     Buffer    ::  hostBuffer(),
80     Why       ::  atom().
81
82 allocHostBuffer(_) -> {error, ecl_lib_not_loaded}.
83
84 %%Allocate a buffer on the device of size SizeByte bytes, specifying kernel read/
       write permissions
85 -spec allocDeviceBuffer(SizeByte, Flags) -> {ok, Buffer} | {error, Why} when
86     SizeByte  ::  non_neg_integer(),
87     Flags     ::  rw_flag(),
88     Buffer    ::  deviceBuffer(),
89     Why       ::  atom().
90
91 allocDeviceBuffer(_,_) -> {error, ecl_lib_not_loaded}.
92
93 %%release a previously allocated buffer
94 -spec releaseBuffer(Buffer) -> ok when
95     Buffer    ::  buffer()
96 .
97
98 releaseBuffer(_) -> {error, ecl_lib_not_loaded}.
99
100 %%Copy data from a buffer to another having the same size
101 -spec copyBufferToBuffer(From, To) -> ok | {error, Why} when
102     From      ::  buffer(),
103     To        ::  buffer(),
104     Why       ::  atom().
105
106 copyBufferToBuffer(_,_) -> {error, ecl_lib_not_loaded}.
107
108 %%Copy CopySizeByte data from a buffer to another
109 -spec copyBufferToBuffer(From, To, CopySizeByte) -> ok | {error, Why} when
110     From      ::  buffer(),
111     To        ::  buffer(),
112     CopySizeByte::  non_neg_integer(),
113     Why       ::  atom().
```

```
114
115 copyBufferToBuffer(_,_,_) -> {error, ecl_lib_not_loaded}.
116
117 %%Copy the content of an Erlang list into a host buffer.
118 -spec listToBuffer(From, To) -> ok | {error, Why} when
119     From     ::  [float()],
120     To       ::  hostBuffer(),
121     Why      ::  atom().
122
123 listToBuffer(_,_) -> {error, ecl_lib_not_loaded}.
124
125
126 %%Create a list having as elements the data in the host buffer
127 -spec bufferToList(From) -> {ok, [float()]} | {error, Why} when
128     From     ::  hostBuffer(),
129     Why      ::  atom().
130
131 bufferToList(_) -> {error, ecl_lib_not_loaded}.
132
133
134 %%Create a list containing the first ListLength elements in the buffer
135 -spec bufferToList(From, ListLength) -> {ok, [float()]} | {error, Why} when
136     From     ::  hostBuffer(),
137     ListLength  ::  pos_integer(),
138     Why      ::  atom().
139
140 bufferToList(_,_) -> {error, ecl_lib_not_loaded}.
141
142 %%Program and Kernel Types and functions ------------------------------
143 -type program()    ::  binary().
144 -type kernel()     ::  binary().
145
146 %%Build an OpenCL program from the source code in the ProgSrcString iolist()
147 -spec  buildProgramFromString(ProgSrcString) -> {ok, program()} | {error, Why} when
148     ProgSrcString:: iolist(),
149     Why      ::  atom().
150
151 buildProgramFromString(_) -> {error, ecl_lib_not_loaded}.
152
153 %%Build an OpenCL program object from the source code in the ProgSrcFile file
154 -spec  buildProgramFromFile(ProgSrcFile) -> {ok, program()} | {error, Why} when
155     ProgSrcFile ::  nonempty_string(),
156     Why      ::  atom().
157
158 buildProgramFromFile(_) -> {error, ecl_lib_not_loaded}.
159
160 %%Create an OpenCL kernel object from a program object.
```

```
161 -spec createKernel(Prog, KerName) -> {ok, kernel()} | {error, Why} when
162     Prog    :: program(),
163     KerName  :: nonempty_string(),
164     Why     :: atom().
165
166 createKernel(_,_) -> {error, ecl_lib_not_loaded}.
167
168
169 %%Map functions ----------------------------
170
171 %%Create a map compatible kernel object from the source in MapSrcFunFile. The kernel
      name is FunName.
172 -spec createMapKernel(MapSrcFunFile, FunName) -> {ok, kernel()} | {error, Why}
    when
173     MapSrcFunFile :: nonempty_string(),
174     FunName     :: nonempty_string(),
175     Why       :: atom().
176
177 createMapKernel(MapSrcFunFile, FunName) ->
178   createMapKernel(MapSrcFunFile, FunName, 1, cache).
179
180 %%Create a map compatible kernel object from the source in MapSrcFunFile. The kernel
      name is FunName, the arity is FunArity
181 -spec createMapKernel(MapSrcFunFile, FunName, FunArity) -> {ok, kernel()} | {error
    , Why} when
182     MapSrcFunFile :: nonempty_string(),
183     FunName     :: nonempty_string(),
184     FunArity    :: pos_integer(),
185     Why       :: atom().
186
187 createMapKernel(MapSrcFunFile, FunName, FunArity) ->
188   createMapKernel(MapSrcFunFile, FunName,FunArity, cache).
189
190
191 %%Create a map compatible kernel object from the source in MapSrcFunFile.
192 %%The kernel name is FunName, the arity is FunArity.
193 %%Program caching policy must also be specified.
194 -spec createMapKernel(MapSrcFunFile, FunName, FunArity, CacheKernel) -> {ok,
    kernel()} | {error, Why} when
195     MapSrcFunFile :: nonempty_string(),
196     FunName     :: nonempty_string(),
197     FunArity    :: pos_integer(),
198     CacheKernel   :: cache | no_cache,
199     Why       :: atom().
200
201 createMapKernel(_,_,_,_,_) -> {error, ecl_lib_not_loaded}.
202
```

```
203 %%Map skeleton working on device buffers. User function specified as kernel objects.
204 -spec mapDD(Kernel, Input, Output) -> ok | {error, Why} when
205     Kernel    :: kernel(),
206     Input     :: deviceBuffer(),
207     Output    :: deviceBuffer(),
208     Why       :: atom().
209
210 mapDD(_,_,_)-> {error, ecl_lib_not_loaded}.
211
212
213 %%Similar to lists:zipWith(Combine, L1, L2) -> L3 but restricted to double
214 %%Binary Map skeleton working on device buffers. User function specified as kernel
        objects.
215 %%Similar to lists:zipWith
216 -spec map2DD(Kernel, Input1, Input2, Output) -> ok | {error, Why} when
217     Kernel    :: kernel(),
218     Input1    :: deviceBuffer(),
219     Input2    :: deviceBuffer(),
220     Output    :: deviceBuffer(),
221     Why       :: atom().
222
223 map2DD(_,_,_,_)-> {error, ecl_lib_not_loaded}.
224
225 %%Map skeleton: input is a list, output is a device buffer. User function specified
        as kernel objects.
226 -spec mapLD(Kernel, Input, Output, InputLength) -> ok | {error, Why} when
227     Kernel    :: kernel(),
228     Input     :: [float()],
229     Output    :: deviceBuffer(),
230     InputLength :: non_neg_integer(),
231     Why       :: atom().
232
233 mapLD(_,_,_,_)-> {error, ecl_lib_not_loaded}.
234
235 %%Binary Map skeleton: input is a list, output is a device buffer. User function
        specified as kernel objects.
236 -spec map2LD(Kernel, Input1, Input2, Output, InputLength) -> ok | {error, Why} when
237     Kernel    :: kernel(),
238     Input1    :: [float()],
239     Input2    :: [float()],
240     Output    :: deviceBuffer(),
241     InputLength :: non_neg_integer(),
242     Why       :: atom().
243
244 map2LD(_,_,_,_,_)-> {error, ecl_lib_not_loaded}.
245
```

```
246 %% Map skeleton: input is a list, output too. User function specified as kernel
        objects.
247 -spec  mapLL(Kernel, Input, InputLength) -> {ok, Result} when
248     Kernel      ::  kernel(),
249     Input       ::  [float()],
250     InputLength ::  non_neg_integer(),
251     Result      ::  [float()].
252
253 mapLL(_,_,_)-> {error, ecl_lib_not_loaded}.
254
255 %%Binary Map skeleton: input is a list, output too. User function specified as
        kernel objects.
256 -spec  map2LL(Kernel, Input1, Input2, InputLength) -> {ok, Result} when
257     Kernel      ::  kernel(),
258     Input1      ::  [float()],
259     Input2      ::  [float()],
260     InputLength ::  non_neg_integer(),
261     Result      ::  [float()].
262
263 map2LL(_,_,_,_,_)-> {error, ecl_lib_not_loaded}.
264
265
266 %%Reduce functions ------------------------------
267
268 %%Create a reduce compatible kernel object from the source in ReduceSrcFunFile.
269 %%The kernel name is FunName.
270 -spec   createReduceKernel(ReduceSrcFunFile, FunName ) -> {ok, kernel()} | {error,
        Why} when
271     ReduceSrcFunFile  ::  nonempty_string(),
272     FunName           ::  nonempty_string(),
273     Why               ::  atom().
274
275 createReduceKernel(ReduceSrcFunFile, FunName) ->
276   createReduceKernel(ReduceSrcFunFile, FunName, cache).
277
278
279 %%Create a reduce compatible kernel object from the source in ReduceSrcFunFile.
280 %%The kernel name is FunName
281 %%Program caching policy must also be specified.
282 -spec   createReduceKernel(ReduceSrcFunFile, FunName, CacheKernel) -> {ok, kernel()}
        | {error, Why} when
283     ReduceSrcFunFile  ::  nonempty_string(),
284     FunName           ::  nonempty_string(),
285     CacheKernel       ::  cache | no_cache,
286     Why               ::  atom().
287
288 createReduceKernel(_,_,_) -> {error, ecl_lib_not_loaded}.
```

```
289
290
291 %% -spec  reduceHH(Kernel, Input) -> ok | {error, Why} when
292 %%     Kernel     ::  kernel(),
293 %%     Input    ::  hostBuffer(),
294 %%     Why      ::  atom().
295 %%
296 %% reduceHH(_,_) -> {error, ecl_lib_not_loaded}.
297
298  %%Reduce skeleton working on device buffers. User function specified as kernel
       objects.
299 -spec  reduceDD(Kernel, Input, Output) -> ok | {error, Why} when
300     Kernel     ::  kernel(),
301     Input    ::  deviceBuffer(),
302     Output     ::  deviceBuffer(),
303     Why      ::  atom().
304
305 reduceDD(_,_,_) -> {error, ecl_lib_not_loaded}.
306
307 %%Reduce skeleton: input is a device buffer, output is a list. User function
       specified as kernel objects.
308 -spec  reduceDL(Kernel, Input) -> {ok, Result} | {error, Why} when
309     Kernel     ::  kernel(),
310     Input    ::  deviceBuffer(),
311     Result     ::  [float()],
312     Why      ::  atom().
313
314 reduceDL(_,_) -> {error, ecl_lib_not_loaded}.
315
316 %%Reduce skeleton: both input and output are lists. User function specified as
       kernel objects.
317 -spec  reduceLL(Kernel, Input, InputLength) -> {ok, Result} | {error, Why} when
318     Kernel     ::  kernel(),
319     Input    ::  [float()],
320     InputLength ::  non_neg_integer(),
321     Result     ::  [float()],
322     Why      ::  atom().
323
324 reduceLL(ReduceKernel, InputList, InputListLength) ->
325
326   SzDouble = 8,
327   SzBufferByte = InputListLength * SzDouble,
328
329   %%allocate host buffer and initialize it using InputList
330   InputH = checkResult( allocHostBuffer(SzBufferByte) ),
331   checkResult(listToBuffer(InputList, InputH)),
332
```

```
333   %%allocate device buffer and copy the input data from the host buffer.
334   InputD = checkResult( allocDeviceBuffer(SzBufferByte, read_write) ),
335   copyBufferToBuffer(InputH, InputD),
336
337   %%don't need host buffer anymore
338   releaseBuffer(InputH),
339
340   %%compute Reduce using the DL version
341   ResultList = reduceDL(ReduceKernel, InputD),
342
343   releaseBuffer(InputD),
344
345   ResultList
346 .
347
348
349 %%Library info
350 skeletonlib(_)->
351    {error, ecl_lib_not_loaded}.
352
353 %%initialize Skel COL
354 cl_init()->
355    {error, ecl_lib_not_loaded}.
356
357 cl_release()->
358   {error, ecl_lib_not_loaded}.
359
360
361 %%------- Skeleton Composition example ------------------
362 %%
363 -spec  mapReduceLL(MapKernel, ReduceKernel, {InputList, InputListLength}) -> {ok,
        Result} when
364     MapKernel ::  kernel(),
365     ReduceKernel::  kernel(),
366     InputList ::  [float()],
367     InputListLength ::  non_neg_integer(),
368     Result    ::  [float()]
369 .
370
371 mapReduceLL(MapKernel, ReduceKernel, {InputList, InputListLength}) ->
372
373   SzDouble = 8,
374   SzBufferByte = InputListLength * SzDouble,
375
376   MapOutputD = checkResult( allocDeviceBuffer(SzBufferByte, read_write) ),
377
378   checkResult( mapLD(MapKernel, InputList, MapOutputD, InputListLength) ),
```

```
379
380   Result = reduceDL(ReduceKernel, MapOutputD),
381
382   releaseBuffer(MapOutputD),
383
384   Result
385 .
386
387 %%Unwrap the Result tuple returning the second element, if the first is an error
      atom throw an exception
388 -spec checkResult(Result) -> Result | Why when
389     Result    :: {ok, Result} | {error, Why},
390     Result    :: any(),
391     Why       :: atom().
392
393 checkResult({ok, Ref}) when is_reference(Ref)->
394   Ref;
395 checkResult({ok, Result}) ->
396   Result;
397 checkResult({error, Why}) ->
398   erlang:error(Why);
399 checkResult(ok) ->
400   ok.
```

B.2 skel_ocl.so

B.2.1 skel_ocl.cpp

```
1 #include "erl_nif.h"
2
3 #include <iostream>
4 #include <fstream>
5 #include <sstream>
6 #include <cstring>
7
8 #include <cerrno>
9
10 #include <vector>
11 #include <unordered_map>
12
13 #include "OCL2.h"
14
15 #include "errors.cpp"
16
17 #include "utils.cpp"
18
19 #include "TimeUtils.h"
20
21 //Kernels' code
22 #include "map_kernel.h"
23 #include "reduce_kernel.h"
24
25 using namespace std;
26
27
28 #define ONE 1
29 #define TWO 2
30
31 const uint NUM_QUEUES = TWO; //ONE or TWO
32
33 const uint MAX_QUEUES = TWO;
34
35
36 static bool ocl_initialised = false;
37
38 #define OCL_INIT_CHECK() if(!ocl_initialised) return make_error(env, ATOM(
      skel_ocl_not_initialized) );
39
40 #define NIF_ARITY_CHECK(ARITY) if(argc != ARITY) return enif_make_badarg(env);
41
42
```

```
43 //defined in OCL2.cpp
44 extern uint device_minBaseAddrAlignByte;
45
46
47
48 /***************  RESOURCE OBJECTS ***************/
49
50 //Define a destructor for OBJTYPE using BODY
51 #define DEF_DTOR(OBJTYPE, BODY) \
52     static void dtor_##OBJTYPE(ErlNifEnv* env, void* obj) { \
53       OBJTYPE* _obj = (OBJTYPE*) obj; \
54       if(*_obj) { \
55                 BODY \
56       } \
57     }
58
59 //Host and device buffers
60 static ErlNifResourceType* hostBuffer_rt = NULL;
61 static ErlNifResourceType* deviceBuffer_rt = NULL;
62
63 DEF_DTOR(cl_mem, (NULL) ;)
64
65 //CL Program
66 static ErlNifResourceType* program_rt = NULL;
67 DEF_DTOR(cl_program, clReleaseProgram(*_obj);)
68
69
70 //CL Kernel with synchronized access
71 static ErlNifResourceType* kernel_sync_rt = NULL;
72
73 typedef struct _kernel_sync {
74   cl_kernel kernel;
75   ErlNifMutex *mtx;
76 } kernel_sync;
77
78 static kernel_sync* ctor_kernel_sync(cl_kernel kernel) {
79
80   kernel_sync* pKernel_s = (kernel_sync*) enif_alloc_resource( kernel_sync_rt,
        sizeof(kernel_sync) );
81
82   pKernel_s->mtx = enif_mutex_create((char*)"setkernelArgMutex");
83
84   pKernel_s->kernel = kernel;
85
86   return pKernel_s;
87 }
88
```

```
89 static void dtor_kernel_sync(ErlNifEnv* env, void* obj)
90 {
91   kernel_sync* ker_s = (kernel_sync*) obj;
92   if(ker_s) {
93
94 #ifdef DEBUG
95     cerr << "DEBUG :: kernel_sync_dtor" << endl;
96 #endif
97     enif_mutex_destroy(ker_s->mtx);
98
99     clReleaseKernel(ker_s->kernel);
100  }
101 }
102
103 //CL event
104 static ErlNifResourceType* event_rt = NULL;
105 DEF_DTOR(cl_event, clReleaseEvent(*_obj);)
106
107 /*
108 * Worker thread handler, exploits NIF garbage collection mechanism
109 * so to be sure to join a thread before module unload (as required by NIF spec)
110 */
111 static ErlNifResourceType* thread_handler_rt = NULL;
112 /*thread handler: it just joins a thread*/
113 DEF_DTOR(ErlNifTid, enif_thread_join(*_obj,NULL);)
114
115
116 //Resource object declaration macro, used in the load function
117 #define DECL_RESOURCE_OBJ(RTYPE_VAR, NAME, DTOR, FLAGS) \
118 { ErlNifResourceType* rt = enif_open_resource_type(env, NULL, NAME, DTOR, FLAGS,
      NULL); \
119   if (rt == NULL) return -1; \
120   RTYPE_VAR = rt;\
121 }
122
123 static int load(ErlNifEnv* env, void** priv_data, ERL_NIF_TERM load_info)
124 {
125
126   /*BUFFERS RO */
127   DECL_RESOURCE_OBJ(hostBuffer_rt, "host_buffer", dtor_cl_mem, ERL_NIF_RT_CREATE)
128   DECL_RESOURCE_OBJ(deviceBuffer_rt, "device_buffer", dtor_cl_mem, ERL_NIF_RT_CREATE
        )
129
130   /*Program and Kernel RO*/
131   DECL_RESOURCE_OBJ(program_rt, "program", dtor_cl_program, ERL_NIF_RT_CREATE)
132   DECL_RESOURCE_OBJ(kernel_sync_rt, "kernel", dtor_kernel_sync, ERL_NIF_RT_CREATE)
133
```

```
134   /*Event RO*/
135   DECL_RESOURCE_OBJ(event_rt, "event", dtor_cl_event, ERL_NIF_RT_CREATE)
136
137   /*worker thread RO*/
138   DECL_RESOURCE_OBJ(thread_handler_rt, "thread_handler", dtor_ErlNifTid,
          ERL_NIF_RT_CREATE)
139
140   return 0;
141 }
142
143
144 /*******************     BUFFER MANAGEMENT NIFs
      ***********************************/
145
146 static const char* MEM_READ_FLAG = "read";
147 static const char* MEM_WRITE_FLAG = "write";
148 static const char* MEM_READ_WRITE_FLAG = "read_write";
149
150 static const size_t MEM_FLAG_ATM_MAX_LEN  = strlen(MEM_READ_WRITE_FLAG)+1;
151
152 static bool parseMemFlags(const char* flag, cl_mem_flags* parsedFlag) {
153
154   //parse flag
155   if(strcmp(MEM_READ_FLAG, flag) == 0)
156     *parsedFlag |= CL_MEM_READ_ONLY;
157   else
158     if(strcmp(MEM_WRITE_FLAG, flag) == 0)
159       *parsedFlag |= CL_MEM_WRITE_ONLY;
160     else
161       if(strcmp(MEM_READ_WRITE_FLAG, flag) == 0)
162         *parsedFlag |= CL_MEM_READ_WRITE;
163       else
164         return false;
165
166   return true;
167 }
168
169
170 static ERL_NIF_TERM getBufferSize(ErlNifEnv * env, int argc, const ERL_NIF_TERM argv
      []) {
171
172   OCL_INIT_CHECK()
173
174   /*get the parameter (Buffer::buffer())***************/
175   NIF_ARITY_CHECK(1)
176
177   cl_mem* pBuffer = NULL;
```

```
178   if (!enif_get_resource(env, argv[0], hostBuffer_rt, (void**) &pBuffer)) { //is it
          a hostBuffer?
179
180     if (!enif_get_resource(env, argv[0], deviceBuffer_rt, (void**) &pBuffer)) { //
            or a deviceBuffer?
181       cerr << "ERROR :: releaseBuffer: 1st parameter is not a buffer" << endl ;
182       return enif_make_badarg(env);
183     }
184   }
185   if(*pBuffer == NULL)
186     return enif_make_badarg(env);
187
188   /*****************************************************/
189
190
191   size_t szBufferByte = 0;
192
193   CHK_SUCCESS(getBufferSizeByte(*pBuffer, &szBufferByte);)
194
195
196   return enif_make_ulong(env, szBufferByte);
197 }
198
199
200 static ERL_NIF_TERM allocDeviceBuffer(ErlNifEnv * env, int argc, const ERL_NIF_TERM
        argv[]) {
201
202   //-type rw_flags() :: read | write | read_write
203   //allocBuffer(SizeByte::non_neg_integer(), Flags::rw_flag()) -> {ok, Buffer} | {
          error, Why}
204
205   OCL_INIT_CHECK()
206
207   /*get the parameter (SizeByte::non_neg_integer(), Flags::rw_flag())**********
          */
208   NIF_ARITY_CHECK(2)
209
210   //SizeByte::non_neg_integer()
211   size_t szBufferByte;
212   if (!enif_get_ulong(env, argv[0], &szBufferByte)) {
213
214     cerr << "ERROR :: allocDeviceBuffer: 1st parameter is not a non_neg_integer()"
            << endl ;
215     return enif_make_badarg(env);
216   }
217
218   //Flags::rw_flag()
```

```
219   char flag[MEM_FLAG_ATM_MAX_LEN];
220   if(! enif_get_atom(env, argv[1], flag, MEM_FLAG_ATM_MAX_LEN, ERL_NIF_LATIN1)) {
221
222     cerr << "ERROR :: allocDeviceBuffer: 2nd parameter is not an rw_flag()" << endl
          ;
223     return enif_make_badarg(env);
224   }
225   /*****************************************************************************
        */
226
227
228   //parse flag
229   cl_mem_flags mem_flags = 0;
230
231   if(!parseMemFlags(flag, &mem_flags)) {
232     cerr << "ERROR :: allocDeviceBuffer: 2nd parameter is not an rw_flag()" << endl
          ;
233     return enif_make_badarg(env);
234   }
235
236   /*allocate the memory for the resource (a cl_mem) */
237   cl_mem* pBuffer = NULL;
238
239   pBuffer = (cl_mem*) enif_alloc_resource(deviceBuffer_rt, sizeof(cl_mem));
240
241   //allocate the buffer
242   CHK_SUCCESS_CLEANUP(createBuffer(szBufferByte, mem_flags, pBuffer); ,
          enif_release_resource(pBuffer);)
243
244   /*grant co-ownership to erlang*/
245   ERL_NIF_TERM bufferRO = enif_make_resource(env, pBuffer);
246
247   /*DON'T transfer the ownership to erlang because a releaseBuffer NIF is provided*/
248
249   return enif_make_tuple2(env, ATOM(ok), bufferRO);
250 }
251
252 static ERL_NIF_TERM allocHostBuffer(ErlNifEnv * env, int argc, const ERL_NIF_TERM
        argv[]) {
253
254   //allocHostBuffer(SizeByte::non_neg_integer()) -> {ok, Buffer} | {error, Why}
255
256   OCL_INIT_CHECK()
257
258   /*get the parameter (SizeByte::non_neg_integer())***************/
259   NIF_ARITY_CHECK(1)
260
```

```
261   //SizeByte::non_neg_integer()
262   size_t szBufferByte;
263   if (!enif_get_ulong(env, argv[0], &szBufferByte)) {
264     cerr << "ERROR :: allocHostBuffer: first parameter is not a non_neg_integer()"
          << endl ;
265     return enif_make_badarg(env);
266   }
267
268   /*************************************************************************
        */
269
270   /*allocate the memory for the resource (a cl_mem) */
271   cl_mem* pBuffer = NULL;
272
273   pBuffer = (cl_mem*) enif_alloc_resource( hostBuffer_rt, sizeof(cl_mem));
274
275   //request buffer in (pinned) host memory
276   CHK_SUCCESS_CLEANUP(createBuffer(szBufferByte, CL_MEM_ALLOC_HOST_PTR, pBuffer); ,
        enif_release_resource(pBuffer);)
277
278   /*grant co-ownership to erlang*/
279   ERL_NIF_TERM bufferRO = enif_make_resource(env, pBuffer);
280
281   /*DON'T transfer the ownership to erlang because a releaseBuffer NIF is provided*/
282
283   return enif_make_tuple2(env, ATOM(ok), bufferRO);
284 }
285
286
287 static ERL_NIF_TERM releaseBuffer(ErlNifEnv * env, int argc, const ERL_NIF_TERM argv
      []) {
288
289   //releaseBuffer(Buffer::buffer())
290
291   OCL_INIT_CHECK()
292
293   /*get the parameter (Buffer::buffer())*********************************/
294   NIF_ARITY_CHECK(1)
295
296   cl_mem* pBuffer = NULL;
297   if (!enif_get_resource(env, argv[0], hostBuffer_rt, (void**) &pBuffer)) { //is it
        a hostBuffer?
298
299     if (!enif_get_resource(env, argv[0], deviceBuffer_rt, (void**) &pBuffer)) { //
          or a deviceBuffer?
300       cerr << "ERROR :: releaseBuffer: 1st parameter is not a buffer" << endl ;
301       return enif_make_badarg(env);
```

```
302     }
303   }
304   if(*pBuffer == NULL)
305     return enif_make_badarg(env);
306 
307   /*****************************************************************************
          */
308 
309 
310   cl_int err =
311       clReleaseMemObject(*pBuffer);
312 
313   *pBuffer = NULL;
314   enif_release_resource(pBuffer);
315 
316   if(err != CL_SUCCESS)
317     return make_error_cl(env,err);
318 
319   return ATOM(ok);
320 
321 }
322 
323 
324 static ERL_NIF_TERM copyBufferToBufferSameSize(ErlNifEnv * env, int argc, const
      ERL_NIF_TERM argv[]) {
325 
326   //copyBufferToBuffer(From::buffer(), To::buffer()) -> ok
327 
328   OCL_INIT_CHECK()
329 
330   /*get the parameters (From::buffer(), To::buffer())**************/
331   NIF_ARITY_CHECK(2)
332 
333   cl_mem* src_buffer = NULL;
334   if (!enif_get_resource(env, argv[0], hostBuffer_rt, (void**) &src_buffer)) { //is
          it a hostBuffer?
335 
336     if (!enif_get_resource(env, argv[0], deviceBuffer_rt, (void**) &src_buffer)) {
            // or a deviceBuffer?
337       cerr << "ERROR :: copyBufferToBufferSameSize: first parameter is not a buffer"
              << endl ;
338       return enif_make_badarg(env);
339     }
340   }
341   if(*src_buffer == NULL)
342     return enif_make_badarg(env);
343 
```

```
344   cl_mem* dst_buffer = NULL;
345   if (!enif_get_resource(env, argv[1], hostBuffer_rt, (void**) &dst_buffer)) { //is
          a hostBuffer?
346
347     if (!enif_get_resource(env, argv[1], deviceBuffer_rt, (void**) &dst_buffer)) {
            // or a deviceBuffer?
348       cerr << "ERROR :: copyBufferToBufferSameSize: 2nd parameter is not a buffer"
              << endl ;
349       return enif_make_badarg(env);
350     }
351   }
352   if(*dst_buffer == NULL)
353       return enif_make_badarg(env);
354
355   /*************************************************************************
          */
356
357
358 #ifdef TIME
359   timespec fun_start, fun_end;
360   GET_TIME((fun_start));
361 #endif
362
363   size_t szSrcBufferByte = 0, szDestBufferByte = 0;
364
365   CHK_SUCCESS(getBufferSizeByte(*src_buffer, &szSrcBufferByte);)
366   CHK_SUCCESS(getBufferSizeByte(*dst_buffer, &szDestBufferByte);)
367
368   // Input Buffers must have the same size
369   if(szSrcBufferByte != szDestBufferByte)
370     return make_error(env, ATOM(skel_ocl_buffers_different_size));
371
372   CHK_SUCCESS(copyBuffer(*src_buffer, *dst_buffer, szSrcBufferByte);)
373
374
375 #ifdef SEQ
376   clFinish(getCommandQueue(0));
377 #endif
378 #ifdef TIME
379   GET_TIME((fun_end));
380
381   cerr << "copy: "<< nsec2usec(diff(&(fun_start), &(fun_end))) << endl;
382 #endif
383
384
385   return ATOM(ok);
386 }
```

```
387
388
389
390 static ERL_NIF_TERM copyBufferToBufferSize(ErlNifEnv * env, int argc, const
        ERL_NIF_TERM argv[]) {
391
392   //copyBufferToBuffer(From::buffer(), To::buffer(), CopySizeByte::non_neg_integer()
          ) -> ok
393
394   OCL_INIT_CHECK()
395
396   /*get the parameters (From::buffer(), To::buffer(), CopySizeByte::non_neg_integer
          ())***************/
397   NIF_ARITY_CHECK(3)
398
399   cl_mem* src_buffer = NULL;
400   if (!enif_get_resource(env, argv[0], hostBuffer_rt, (void**) &src_buffer)) { //is
          it a hostBuffer?
401
402     if (!enif_get_resource(env, argv[0], deviceBuffer_rt, (void**) &src_buffer)) {
            // or a deviceBuffer?
403       cerr << "ERROR :: copyBufferToBufferSize: first parameter is not a buffer" <<
              endl ;
404       return enif_make_badarg(env);
405     }
406   }
407   if(*src_buffer == NULL)
408     return enif_make_badarg(env);
409
410   cl_mem* dst_buffer = NULL;
411   if (!enif_get_resource(env, argv[1], hostBuffer_rt, (void**) &dst_buffer)) { //is
          a hostBuffer?
412
413     if (!enif_get_resource(env, argv[1], deviceBuffer_rt, (void**) &dst_buffer)) {
            // or a deviceBuffer?
414       cerr << "ERROR :: copyBufferToBufferSize: 2nd parameter is not a buffer" <<
              endl ;
415       return enif_make_badarg(env);
416     }
417   }
418   if(*dst_buffer == NULL)
419       return enif_make_badarg(env);
420
421   //CopySizeByte::non_neg_integer()
422   size_t szCopyByte;
423   if (!enif_get_ulong(env, argv[2], &szCopyByte)) {
424
```

```
425     cerr << "ERROR :: copyBufferToBuffer: 3rd parameter is not a non_neg_integer()"
            << endl ;
426     return enif_make_badarg(env);
427   }
428
429   /*****************************************************************************
          */
430
431   CHK_SUCCESS(copyBuffer(*src_buffer, *dst_buffer, szCopyByte);)
432
433   return ATOM(ok);
434 }
435
436
437
438 /** Copies the content of a list into a buffer*/
439 static ERL_NIF_TERM listToBuffer(ErlNifEnv * env, int argc, const ERL_NIF_TERM argv
        []) {
440
441   //listToBuffer(From::[float()], To::hostBuffer()) -> ok | {error, Why}
442   OCL_INIT_CHECK()
443
444   /*get the parameters (To::list())***************/
445   NIF_ARITY_CHECK(2)
446
447   ERL_NIF_TERM list = argv[0];
448   if(!enif_is_list(env, list)) {
449     cerr << "ERROR :: listToBuffer: first parameter is not a list" << endl;
450     return enif_make_badarg(env);
451   }
452
453   cl_mem* pBuffer = NULL;
454   if (!enif_get_resource(env, argv[1], hostBuffer_rt, (void**) &pBuffer) || *pBuffer
          == NULL) {
455     cerr << "ERROR :: listToBuffer: second parameter is not a buffer" << endl ;
456     return enif_make_badarg(env);
457   }
458   /*********************************************************/
459
460 #ifdef TIME
461   timespec fun_start, fun_end;
462   GET_TIME((fun_start));
463 #endif
464
465   //copy
466   unsigned int list_len = 0, szListBytes = 0;
467
```

```
468   enif_get_list_length(env, list, &list_len);
469   szListBytes = list_len * sizeof(double);
470
471
472   double* pMappedBuffer = NULL;
473   //obtain a pointer to work on the buffer (out of bounds check done by OpenCL,
          returns CL_INVALID_VALUE)
474   CHK_SUCCESS( mapBufferBlocking(*pBuffer, 0, szListBytes, CL_MAP_WRITE, &
          pMappedBuffer); )
475
476   if(pMappedBuffer == NULL) {
477     cerr << "pMappedBuffer is NULL."<< endl;
478     return make_error(env, ATOM(cl_error));
479   }
480
481
482   // copies the content of input list into pBuffer
483   list_to_double_array(env, list, pMappedBuffer);
484
485   //unMap when done copying
486   CHK_SUCCESS( unMapBuffer(*pBuffer, pMappedBuffer); )
487
488
489 #ifdef TIME
490   GET_TIME((fun_end));
491
492   cerr << "Unmarshalling: "<< nsec2usec(diff(&(fun_start), &(fun_end))) << endl;
493 #endif
494
495   return ATOM(ok);
496 }
497
498 /** Create a list containing the elements of a buffer*/
499 static ERL_NIF_TERM _bufferToListLength(ErlNifEnv * env, cl_mem* pBuffer, size_t
          listLen) {
500
501   size_t listLenByte = listLen * sizeof(double);
502
503   double* pDoubleArray = NULL;
504
505   //obtain a pointer to work on the buffer
506   CHK_SUCCESS( mapBufferBlocking(*pBuffer, 0, listLenByte, CL_MAP_READ, &
          pDoubleArray);)
507
508   ERL_NIF_TERM list = double_array_to_list(env, pDoubleArray, listLen);
509
510   //unMap when done
```

```
511   CHK_SUCCESS( unMapBuffer(*pBuffer, pDoubleArray); )
512
513   return enif_make_tuple2(env, ATOM(ok), list);
514
515 }
516
517
518 /** Create a list containing the elements of a buffer*/
519 static ERL_NIF_TERM bufferToListLength(ErlNifEnv * env, int argc, const ERL_NIF_TERM
        argv[]) {
520
521   //%%bufferToList(From::buffer(), Length::pos_integer()) -> {ok, [float()]}
522
523   OCL_INIT_CHECK()
524
525   /*get the parameters (From::Buffer, Length::pos_integer())***************/
526   NIF_ARITY_CHECK(2)
527
528   cl_mem* pBuffer = NULL;
529   if (!enif_get_resource(env, argv[0], hostBuffer_rt, (void**) &pBuffer) || *pBuffer
          == NULL) {
530
531     cerr << "ERROR :: bufferToList: 1st parameter is not a host buffer" << endl ;
532     return enif_make_badarg(env);
533   }
534
535   //Length::pos_integer()
536   size_t listLen;
537   if (!enif_get_ulong(env, argv[1], &listLen)) {
538
539     cerr << "ERROR :: bufferToList: 2nd parameter is not a non_neg_integer()" <<
           endl ;
540     return enif_make_badarg(env);
541   }
542
543   /*********************************************/
544
545   // check that buffer contains enough elements
546
547   size_t szBufferByte = 0;
548   CHK_SUCCESS( getBufferSizeByte(*pBuffer, &szBufferByte); )
549
550   size_t listLenByte = listLen * sizeof(double);
551
552   if(listLenByte > szBufferByte) {
553     cerr << "ERROR :: bufferToList: buffer doesn't contain Length elements" << endl
          ;
```

```
554     return enif_make_badarg(env);
555   }
556
557   return _bufferToListLength(env, pBuffer, listLen);
558 }
559
560 /** Create a list containing the elements of a buffer*/
561 static ERL_NIF_TERM bufferToList(ErlNifEnv * env, int argc, const ERL_NIF_TERM argv
        []) {
562
563   //bufferToList(From::Buffer) -> {ok, []} | {error, Why}
564   OCL_INIT_CHECK()
565   /*get the parameter (From::Buffer)***************/
566
567   NIF_ARITY_CHECK(1)
568
569   cl_mem* pBuffer = NULL;
570   if (!enif_get_resource(env, argv[0], hostBuffer_rt, (void**) &pBuffer) || *pBuffer
          == NULL) {
571
572     cerr << "ERROR :: bufferToList: first parameter is not a host buffer" << endl ;
573     return enif_make_badarg(env);
574   }
575   /*********************************************/
576
577
578 #ifdef TIME
579   timespec fun_start, fun_end;
580   GET_TIME((fun_start));
581 #endif
582
583   size_t szBufferByte;
584
585   CHK_SUCCESS( getBufferSizeByte(*pBuffer, &szBufferByte); )
586
587   size_t numBufferElements = szBufferByte / sizeof(double);
588
589   ERL_NIF_TERM toReturn = _bufferToListLength(env, pBuffer, numBufferElements);
590
591
592 #ifdef TIME
593   GET_TIME((fun_end));
594
595   cerr << "Marshalling: "<< nsec2usec(diff(&(fun_start), &(fun_end))) << endl;
596 #endif
597
598   return toReturn;
```

```
599 }
600
601
602
603
604 /********************     PROGRAM AND KERNEL NIFS     *************************/
605
606 static ERL_NIF_TERM buildProgramFromCharP(ErlNifEnv * env, const char* progSrc){
607
608   /*allocate the memory for the resource (a cl_program) */
609   cl_program* pProgram = NULL;
610
611   pProgram = (cl_program*) enif_alloc_resource(
612               program_rt,
613               sizeof(cl_program)
614             );
615
616   //create and build the program releasing the resource in case of errors
617   CHK_SUCCESS_CLEANUP(createBuildProgramFromString(progSrc, pProgram); ,
        enif_release_resource(pProgram);)
618
619 //  enif_free(progSrc); should be freed by the caller
620
621   /*grant co-ownership to erlang*/
622   ERL_NIF_TERM programRO = enif_make_resource(env, pProgram);
623
624   /*transfer the ownership to erlang*/
625   enif_release_resource(pProgram);
626
627   return enif_make_tuple2(env, ATOM(ok), programRO);
628
629 }
630
631 //Unmarshal an Erlang list allocating the needed space
632 #define ENIF_GET_STRING(CHAR_P, ARG) \
633 { \
634     unsigned int strLen = 0; /*without \0 */ \
635     if(!enif_get_list_length(env, ARG, &strLen)) \
636       return enif_make_badarg(env); \
637     CHAR_P = (char *) enif_alloc(strLen++); /*including null terminator*/\
638     enif_get_string(env, ARG, CHAR_P, strLen, ERL_NIF_LATIN1); \
639 }
640
641 #define ENIF_GET_ATOM(CHAR_P, ARG) \
642 { \
643   uint atom_len = 0; \
644   if (!enif_get_atom_length(env, ARG,&atom_len, ERL_NIF_LATIN1)) \
```

```
645     return enif_make_badarg(env); \
646   CHAR_P = (char *) enif_alloc(atom_len++); /*including null terminator*/ \
647   enif_get_atom(env, ARG, CHAR_P, atom_len,ERL_NIF_LATIN1); \
648 }
649
650 static ERL_NIF_TERM buildProgramFromString(ErlNifEnv * env, int argc, const
      ERL_NIF_TERM argv[]) {
651
652   //buildProgramFromString(ProgramSrcString::iolist())
653   OCL_INIT_CHECK()
654   /*get the parameter (ProgramSrcString::iolist())*********************************
        */
655   NIF_ARITY_CHECK(1)
656
657   //ProgramSrcString may be a binary
658   char *progSrc = NULL;
659   ErlNifBinary bin;
660
661   if(enif_is_binary(env, argv[0])) {
662
663     enif_inspect_iolist_as_binary(env, argv[0], &bin);
664
665     progSrc = (char*) enif_alloc(bin.size+1);
666
667     strncpy(progSrc, (char*) bin.data, bin.size);
668
669   }
670   else //ProgramSrcString must then be a list
671     ENIF_GET_STRING(progSrc, argv[0])
672
673   /*****************************************************************************
        */
674
675   ERL_NIF_TERM toReturnTerm = buildProgramFromCharP(env, progSrc);
676
677   if(!progSrc)
678     enif_free(progSrc);
679
680   return toReturnTerm;
681
682 }
683
684 static ERL_NIF_TERM buildProgramFromFile(ErlNifEnv * env, int argc, const
      ERL_NIF_TERM argv[]) {
685
686   //buildProgram(ProgramSrcPath::nonempty_string())
687   OCL_INIT_CHECK()
```

```
688   /*get the parameter (ProgramSrcPath::nonempty_string())
          ***************************/
689   NIF_ARITY_CHECK(1)
690
691   char *progPath = NULL;
692
693   ENIF_GET_STRING(progPath, argv[0]) // progPath must be enif_free'd
694
695   /*************************************************************************
          */
696
697   //Read from the file
698   char* progSrc = NULL;
699
700   if(! (progSrc = readFromFile(progPath)) )
701     return enif_make_tuple2(env, ATOM(error), enif_make_atom(env,"opening_file"));
702
703
704   ERL_NIF_TERM toReturn = buildProgramFromCharP(env, progSrc);
705
706   enif_free(progPath);
707   enif_free(progSrc);
708
709   return toReturn;
710
711 }
712
713 static ERL_NIF_TERM _createKernel(ErlNifEnv * env, cl_program* pProgram, const char*
          kerName) {
714
715   cl_kernel kernel = NULL;
716   CHK_SUCCESS(createKernel(*pProgram, kerName, &kernel); )
717
718   /*crate a kernel_sync*/
719   kernel_sync* pKernel_s = ctor_kernel_sync(kernel);
720
721   /*grant co-ownership to erlang*/
722   ERL_NIF_TERM kernelRO = enif_make_resource(env, pKernel_s);
723
724   /*transfer the ownership to erlang*/
725   enif_release_resource(pKernel_s);
726
727
728   return enif_make_tuple2(env, ATOM(ok), kernelRO);
729 }
730
```

```
731 static ERL_NIF_TERM createKernel(ErlNifEnv * env, int argc, const ERL_NIF_TERM argv
        []) {
732
733   //%%createKernel(Prog::program(), KerName::nonempty_string())
734   OCL_INIT_CHECK()
735   /*get the parameters (Prog::program(), KerName::nonempty_string())**************/
736   NIF_ARITY_CHECK(2)
737
738   //get Program
739
740   cl_program* pProgram = NULL;
741   if (!enif_get_resource(env, argv[0], program_rt, (void**) &pProgram)) {
742       cerr << "ERROR :: createKernel: 1st parameter is not a program" << endl ;
743       return enif_make_badarg(env);
744   }
745
746   char* kerName = NULL;
747   ENIF_GET_STRING(kerName, argv[1])
748
749   /*****************************************************************************
        */
750
751   ERL_NIF_TERM toReturn = _createKernel(env, pProgram, kerName);
752
753   enif_free(kerName);
754
755   return toReturn;
756
757 }
758
759
760 static ERL_NIF_TERM generateProgramSrcFromFile(ErlNifEnv * env, char* funSrcPath,
      string fun_name, string type, string mapKernelStr, string& progSrcStr) {
761
762     string funSrcStr = readFromFileStr(funSrcPath);
763
764     if( funSrcStr.compare("NULL") == 0)
765       return enif_make_tuple2(env, ATOM(error), ATOM(opening_file));
766
767     string kernelSrcStr(mapKernelStr);
768
769     replaceTextInString(kernelSrcStr, std::string("TYPE"), type);
770     replaceTextInString(kernelSrcStr, std::string("FUN_NAME"), fun_name);
771
772     //generate the final program from the src from file and kernel definition
773
774     if(type.compare("double") == 0)
```

```
775       progSrcStr.append("#pragma OPENCL EXTENSION cl_khr_fp64: enable\n");
776
777     progSrcStr.append(funSrcStr);
778     progSrcStr.append(kernelSrcStr);
779
780     return ATOM(ok);
781
782 }
783
784
785 typedef unordered_map<std::string, cl_program> CLPrograms_Cache;
786
787 //Store built programs so to avoid building them again when used more than once
788 static CLPrograms_Cache* clPrograms_Cache;
789
790
791 static ERL_NIF_TERM createSkeletonKernelFromFile(ErlNifEnv * env, char*
        skeletonFunSrcFilePath, const string& skeletonFunName, const string&
        skeletonSrcStr, const char* kernelName, const string& type, bool cache_program)
792 {
793
794   cl_program program;
795
796   string skeletonFunSrcPath_str = string(skeletonFunSrcFilePath);
797
798   if(cache_program && (clPrograms_Cache->count(skeletonFunSrcPath_str) > 0)) { //1
          found, 0 not found
799
800     #ifdef DEBUG
801     cerr<< "DEBUG - looking for cached program.\n";
802     #endif
803     program = (*clPrograms_Cache)[skeletonFunSrcPath_str]; //use cached program
804
805   }
806   else {
807
808     #ifdef DEBUG
809     cerr<< "DEBUG - generating program.\n";
810     #endif
811     //generate the source for the final program from the source code in the file and
            skeleton kernel definition
812     string progSrcStr;
813     ERL_NIF_TERM result_atom =
814         generateProgramSrcFromFile(env, skeletonFunSrcFilePath, skeletonFunName,
              type, skeletonSrcStr, progSrcStr);
815
816     if(result_atom != ATOM(ok))
```

```
817       return result_atom;
818
819     const char* progSrc =  progSrcStr.c_str();
820
821     CHK_SUCCESS(createBuildProgramFromString(progSrc, &program);) // program never
            released
822
823     if(cache_program) {
824       #ifdef DEBUG
825       cerr<< "DEBUG - adding new program to cache.\n";
826       #endif
827       (*clPrograms_Cache)[skeletonFunSrcPath_str] = program; //save built program
              into cache
828     }
829
830   }
831
832   return _createKernel(env, &program, kernelName);
833
834
835 }
836
837 static ERL_NIF_TERM createMapKernelFromFile(ErlNifEnv * env, int argc, const
      ERL_NIF_TERM argv[]) {
838
839   //createMapKernel(MapSrcFunFile::nonempty_string(), FunName::nonempty_string(),
          FunArity::pos_integer(), ProgCaching::atom())
840   OCL_INIT_CHECK()
841   /*get the parameter (MapFunSrcPath::nonempty_string())**************************
          */
842   NIF_ARITY_CHECK(4)
843
844   char *mapFunSrcPath = NULL;
845   ENIF_GET_STRING(mapFunSrcPath, argv[0]) // mapFunSrcPath must be enif_free'd
846
847   char *mapFunName = NULL;
848   ENIF_GET_STRING(mapFunName, argv[1]) // mapFunName must be enif_free'd
849
850   uint funArity = 0;
851   enif_get_uint(env,argv[2], &funArity);
852
853   if(! enif_is_atom(env,argv[3]))
854     return enif_make_badarg(env);
855
856   ERL_NIF_TERM prog_caching_atom = argv[3];
857
```

```
858   /*************************************************************************
          */
859
860
861   const string* pSkeletonSrcStr;
862   const char* mapKernelName;
863
864   switch (funArity)
865   { case 1 :  mapKernelName = "MapKernel";
866           pSkeletonSrcStr = &MapKernelStr;
867           break;
868
869     case 2 :  mapKernelName = "MapKernel2";
870           pSkeletonSrcStr = &Map2KernelStr;
871           break;
872
873     default:  enif_free(mapFunSrcPath); //FunArity not supported
874           enif_free(mapFunName);
875           return make_error(env, ATOM(skel_ocl_fun_arity_not_supported));
876   }
877
878   const char* type = "double";
879
880   bool cache_program = enif_is_identical(ATOM(cache), prog_caching_atom);
881
882   ERL_NIF_TERM toReturn =
883       createSkeletonKernelFromFile(env, mapFunSrcPath, string(mapFunName), *
              pSkeletonSrcStr, mapKernelName, type, cache_program);
884
885   enif_free(mapFunSrcPath);
886   enif_free(mapFunName);
887
888   return toReturn;
889
890 }
891
892 static ERL_NIF_TERM createReduceKernelFromFile(ErlNifEnv * env, int argc, const
      ERL_NIF_TERM argv[]) {
893
894   //buildProgram(ReduceFunSrcPath::nonempty_string(), ReduceFunName::nonempty_string
          ())
895   OCL_INIT_CHECK()
896   /*get the parameter (ReduceFunSrcPath::nonempty_string())*********************
          */
897   NIF_ARITY_CHECK(3)
898
899   char *reduceFunSrcPath = NULL;
```

```
900
901   ENIF_GET_STRING(reduceFunSrcPath, argv[0]) // reduceFunSrcPath must be enif_free'd
902
903   char *reduceFunName = NULL;
904
905   ENIF_GET_STRING(reduceFunName, argv[1]) // reduceFunName must be enif_free'd
906
907
908   if(! enif_is_atom(env,argv[2]))
909     return enif_make_badarg(env);
910
911   ERL_NIF_TERM prog_caching_atom = argv[2];
912
913   /*****************************************************************************
          */
914
915
916   const char* reduceKernelName = "ReduceKernel";
917   const char* type = "double";
918
919   bool cache_program = enif_is_identical(ATOM(cache), prog_caching_atom);
920
921   ERL_NIF_TERM toReturn =
922       createSkeletonKernelFromFile(env, reduceFunSrcPath, string(reduceFunName),
              ReduceKernelStr, reduceKernelName, type, cache_program);
923
924   enif_free(reduceFunSrcPath);
925   enif_free(reduceFunName);
926
927   return toReturn;
928
929 }
930
931 /*****************    SKELETONS    ***********************************************/
932
933
934 #include "map_nifs.cpp"
935
936 #include "reduce_nifs.cpp"
937
938
939
940
941
942
943 /*****************    MISCs    ***************************************************/
944
```

```
945 /**
946  * Initialize OpenCL.
947  * Returns
948  * - ok
949  * - {error, "OpenCL already initialized."} when called more than once
950  * - badarg in the case of any error during the initialization
951  */
952 static ERL_NIF_TERM initOCL(ErlNifEnv * env, int argc, const ERL_NIF_TERM argv[]) {
953
954   if(!ocl_initialised) {
955
956     if(!init())
957       return make_error(env, ATOM(openCL_init_failed) );
958     else {
959       ocl_initialised = true;
960
961       //init cl_program cache
962       clPrograms_Cache = new CLPrograms_Cache();
963
964       return ATOM(ok);
965     }
966   }
967   else return
968       make_error(env, ATOM(openCL_already_init) );
969 }
970
971 static ERL_NIF_TERM releaseOCL(ErlNifEnv * env, int argc, const ERL_NIF_TERM argv[])
      {
972
973   if(ocl_initialised) {
974
975 #ifdef DEBUG
976     std::cerr << "calling releaseOCL()" << std::endl;
977 #endif
978     releaseOCL();
979 #ifdef DEBUG
980     std::cerr << "releaseOCL() ok" << std::endl;
981 #endif
982
983     delete clPrograms_Cache;
984
985     ocl_initialised = false;
986
987     return ATOM(ok);
988   }
989   else return enif_make_tuple2(
990       env,
```

```
991       ATOM(error),
992       ATOM(OpenCL_not_initialized)
993     );
994
995 }
996
997 /* NIF interface bindings */
998 static ErlNifFunc nif_funcs[] = {
999
1000     {"skeletonlib"       , 1,  skeletonlib       },
1001     {"cl_init"           , 0,  initOCL           },
1002     {"cl_release"        , 0,  releaseOCL        },
1003
1004     //Buffer
1005     {"listToBuffer"      , 2,  listToBuffer      },
1006     {"bufferToList"      , 1,  bufferToList      },
1007     {"bufferToList"      , 2,  bufferToListLength    },
1008     {"getBufferSize"     , 1,  getBufferSize     },
1009     {"allocDeviceBuffer" , 2,  allocDeviceBuffer   },
1010     {"allocHostBuffer"   , 1,  allocHostBuffer   },
1011     {"releaseBuffer"     , 1,  releaseBuffer     },
1012     {"copyBufferToBuffer" , 2,  copyBufferToBufferSameSize},
1013     {"copyBufferToBuffer" , 3,  copyBufferToBufferSize  },
1014
1015     //Program and kernel
1016     {"buildProgramFromFile" , 1,  buildProgramFromFile  },
1017     {"buildProgramFromString",1,  buildProgramFromString  },
1018     {"createKernel"      , 2,  createKernel      },
1019
1020     //MAP
1021     {"createMapKernel"     , 4,  createMapKernelFromFile },
1022     {"mapDD"             , 3,  mapDD           },
1023     {"mapLD"             , 4,  mapLD           },
1024     {"mapLL"             , 3,  mapLL           },
1025
1026     {"map2DD"            , 4,  map2DD            },
1027     {"map2LD"            , 5,  map2LD            },
1028     {"map2LL"            , 4,  map2LL            },
1029
1030     //REDUCE
1031     {"createReduceKernel" , 3,  createReduceKernelFromFile},
1032     {"reduceDD"          , 3,  reduceDD          },
1033     {"reduceDL"          , 2,  reduceDL          }
1034
1035 };
1036
1037 ERL_NIF_INIT(skel_ocl,nif_funcs,load,NULL,NULL,NULL)
```

B.2.2 Map NIFs: `map_nifs.cpp`

```
1  /*****************   MAP   ********************************************/
2
3
4  /***************Map on device buffers**********************/
5
6  /*Implementation of n-ary map using device buffers */
7  /* Requirements:
8   * - All buffers in pInputBuffrV must have the same size,
9   * - pInputBufferC > 0
10  * */
11 static ERL_NIF_TERM mapDD_Impl(ErlNifEnv * env, kernel_sync* pKernel_s, void**
       _pInputBufferV, uint pInputBufferC, void* _pOutputBuffer) {
12
13 #ifdef TIME
14     timespec
15     fun_start,
16     fun_prologue_end,
17     run_end,
18     fun_end;
19
20     GET_TIME((fun_start));
21 #endif
22
23     cl_mem** pInputBufferV  = (cl_mem**)_pInputBufferV;
24     cl_mem* pOutputBuffer = (cl_mem*) _pOutputBuffer;
25
26     const uint OFFSET = 0;
27
28     //All buffers have the same size, then check just the first one
29     size_t szInputBufferByte = 0;
30     CHK_SUCCESS(getBufferSizeByte(*(pInputBufferV[0]), &szInputBufferByte);)
31
32     //**********set map kernel parameters**********************
33     uint uiNumElem = szInputBufferByte / sizeof(double);
34
35     cl_int ciErrNum = 0;
36
37     uint i = 0;
38
39     // clSetKernelArg is not thread-safe
40     enif_mutex_lock(pKernel_s->mtx);
41
42     for(i=0; i < pInputBufferC; i++)
43         ciErrNum |= clSetKernelArg(pKernel_s->kernel, i, sizeof(cl_mem), (void*)
               pInputBufferV[i]);
```

```
44
45      i--;
46      ciErrNum |= clSetKernelArg(pKernel_s->kernel, i+1, sizeof(cl_mem), (void*)
            pOutputBuffer);
47      ciErrNum |= clSetKernelArg(pKernel_s->kernel, i+2, sizeof(unsigned int), (void*)
            &OFFSET); //Offset
48      ciErrNum |= clSetKernelArg(pKernel_s->kernel, i+3, sizeof(unsigned int), (void*)
            &uiNumElem);
49
50      enif_mutex_unlock(pKernel_s->mtx);
51
52      CHK_SUCCESS(ciErrNum;)
53
54
55 #ifdef TIME
56      GET_TIME((fun_prologue_end));
57 #endif
58
59      //**********execute the kernel*********************
60
61      size_t szGlobalWorkSize =  uiNumElem;
62      //  size_t szLocalWorkSize =  1;
63
64      //  szGlobalWorkSize = roundUp((int)szLocalWorkSize, (int)(szGlobalWorkSize));
65
66      //esegui calcolo
67      CHK_SUCCESS(computeKernel(pKernel_s->kernel, &szGlobalWorkSize, NULL/*&
            szLocalWorkSize*/);) //let OpenCL decide the local work-size
68
69 #ifdef TIME
70      GET_TIME((run_end));
71
72      fun_end = run_end;
73 #endif
74
75 #ifdef TIME
76      cerr <<
77      "prologue + kernel time: " << nsec2usec(diff(&(fun_start), &(fun_end))) <<
78      endl <<
79      "\tFunction prologue time: " << nsec2usec(diff(&(fun_start), &fun_prologue_end))
            <<
80      endl <<
81      "\tKERNEL computation time: " << nsec2usec(diff(&(fun_prologue_end), &run_end))
            <<
82      endl
83      ;
84 #endif
```

```
85
86      return ATOM(ok);
87
88  }
89
90  /* unary map adapter (in/out on device)
91   *It just interprets erlang NIF parameters and calls mapDD_Impl
92   **/
93  static ERL_NIF_TERM mapDD(ErlNifEnv * env, int argc, const ERL_NIF_TERM argv[]) {
94
95      //mapDD(Kernel::kernel(), InputBuffer::deviceBuffer(), OutputBuffer::
            deviceBuffer())
96      OCL_INIT_CHECK()
97      NIF_ARITY_CHECK(3)
98
99      /*get the parameters (Kernel::kernel(), InputBuffer::deviceBuffer(),
            OutputBuffer::deviceBuffer())***********/
100
101     kernel_sync* pKernel_s = NULL;
102     if (!enif_get_resource(env, argv[0], kernel_sync_rt, (void**) &pKernel_s)) {
103         cerr << "ERROR :: mapDD: 1st parameter is not a kernel_sync" << endl ;
104         return enif_make_badarg(env);
105     }
106
107     cl_mem* pInputBuffer = NULL;
108     if (!enif_get_resource(env, argv[1], deviceBuffer_rt, (void**) &pInputBuffer) ||
             *pInputBuffer == NULL) {
109         cerr << "ERROR :: mapDD: 2nd parameter is not a device buffer" << endl ;
110         return enif_make_badarg(env);
111     }
112
113     cl_mem* pOutputBuffer = NULL;
114     if (!enif_get_resource(env, argv[2], deviceBuffer_rt, (void**) &pOutputBuffer)
            || *pOutputBuffer == NULL) {
115         cerr << "ERROR :: mapDD: 3nd parameter is not a device buffer" << endl ;
116         return enif_make_badarg(env);
117     }
118     /*************************************************************************
            */
119
120     uint pInputBufferC = 1;
121     cl_mem* pInputBufferV[1] = { pInputBuffer };
122
123     return mapDD_Impl(env, pKernel_s,(void**) pInputBufferV, pInputBufferC,
            pOutputBuffer);
124 }
125
```

```
126
127
128 /*Binary map adapter (in/out on device)
129  *It just interprets erlang NIF parameters and calls mapDD_impl
130  **/
131 static ERL_NIF_TERM map2DD(ErlNifEnv * env, int argc, const ERL_NIF_TERM argv[]) {
132
133     //mapDD(Kernel::kernel(), InputBuffer1::deviceBuffer(), InputBuffer2::
            deviceBuffer(), OutputBuffer::deviceBuffer())
134     OCL_INIT_CHECK()
135     NIF_ARITY_CHECK(4)
136
137     /*get the parameters (Kernel::kernel(), InputBuffer::deviceBuffer(),
            OutputBuffer::deviceBuffer())***********/
138
139     kernel_sync* pKernel_s = NULL;
140     if (!enif_get_resource(env, argv[0], kernel_sync_rt, (void**) &pKernel_s)) {
141         cerr << "ERROR :: map2DD: 1st parameter is not a kernel_sync" << endl ;
142         return enif_make_badarg(env);
143     }
144
145     cl_mem* pInputBuffer1 = NULL;
146     if (!enif_get_resource(env, argv[1], deviceBuffer_rt, (void**) &pInputBuffer1)
            || *pInputBuffer1 == NULL) {
147         cerr << "ERROR :: map2DD: 2nd parameter is not a device buffer" << endl ;
148         return enif_make_badarg(env);
149     }
150
151     cl_mem* pInputBuffer2 = NULL;
152     if (!enif_get_resource(env, argv[2], deviceBuffer_rt, (void**) &pInputBuffer2)
            || *pInputBuffer2 == NULL) {
153         cerr << "ERROR :: map2DD: 3nd parameter is not a device buffer" << endl ;
154         return enif_make_badarg(env);
155     }
156
157     cl_mem* pOutputBuffer = NULL;
158     if (!enif_get_resource(env, argv[3], deviceBuffer_rt, (void**) &pOutputBuffer)
            || *pOutputBuffer == NULL) {
159         cerr << "ERROR :: map2DD: 4th parameter is not a device buffer" << endl ;
160         return enif_make_badarg(env);
161     }
162     /*************************************************************************
            */
163
164     size_t szInputBuffer1Byte = 0, szInputBuffer2Byte = 0;
165
166     CHK_SUCCESS(getBufferSizeByte(*pInputBuffer1, &szInputBuffer1Byte);)
```

```
167     CHK_SUCCESS(getBufferSizeByte(*pInputBuffer2, &szInputBuffer2Byte);)
168
169     // Input Buffers must have the same size
170     if(szInputBuffer1Byte != szInputBuffer2Byte)
171         return make_error(env, ATOM(skel_ocl_buffers_different_size));
172
173     uint pInputBufferC = 2; //Map2
174     cl_mem * pInputBufferV[2] = { pInputBuffer1, pInputBuffer2 };
175
176     return mapDD_Impl(env,pKernel_s,(void**) pInputBufferV, pInputBufferC,
            pOutputBuffer);
177
178 }
179
180 //counters struct for map profiling
181 struct Map_fun_counters {
182
183     struct Map_fun_round_counters  {
184         uint numInput;
185         timespec
186         start[2],
187         load_start[2],
188         *load_input_start[2],
189         *load_unmarsh_end[2],
190         *load_copyHD_end[2],
191         load_end[2],
192
193         run_start[2],
194         run_end[2],
195
196         unload_start,
197         unload_copy_start[2],
198         unload_copy_end[2],
199         unload_marsh_end[2],
200         unload_end,
201         end[2];
202
203
204         Map_fun_round_counters(uint _numInput) {
205
206             numInput = _numInput;
207
208             for(uint i = 0; i< 2; i++) {
209                 load_input_start[i] = new timespec[numInput];
210                 load_unmarsh_end[i] = new timespec[numInput];
211                 load_copyHD_end[i] = new timespec[numInput];
212             }
```

```
213
214
215         };
216
217         ~Map_fun_round_counters(){
218
219             for(uint i = 0; i< 2; i++) {
220                 delete [] load_input_start[i];
221                 delete [] load_unmarsh_end[i];
222                 delete [] load_copyHD_end[i];
223             }
224
225         };
226
227
228     }; //end Map_fun_round_counters
229
230     timespec
231     fun_start, fun_prologue, fun_end,
232     mapInput_start, *mapInput_end,
233     unmapInput_start, unmapInput_end,
234     mapOutput_start, mapOutput_end,
235     unmapOutput_end,
236
237     releaseInBuffersH_T, releaseOutBufferH_T,
238     releaseOutBufferD_T, releaseInBuffersD_T_start, releaseInBuffersD_T_end
239     ;
240
241     uint numRounds;
242
243     Map_fun_round_counters** round;
244
245     Map_fun_counters(const uint& _numRounds, const uint& numInput) {
246
247         mapInput_end = new timespec[numInput];
248
249         numRounds = _numRounds;
250
251         round = new Map_fun_round_counters*[numRounds];
252
253         for(uint i = 0; i< numRounds; i++)
254             round[i] = new Map_fun_round_counters(numInput);
255     }
256
257     ~Map_fun_counters() {
258
259
```

```
260         for(uint i = 0; i < numRounds; i++)
261             delete round[i];
262 
263         delete [] round;
264 
265         delete [] mapInput_end;
266 
267     }
268 
269 };
270 
271 //Struct containing data shared among mapLL's threads
272 class MapLL_thread_params{
273 
274 public:
275     bool errorSignal;
276     bool requestedExit;
277 
278     ERL_NIF_TERM errorTerm;
279 
280     cl_event mapOutput_evt;
281     cl_event unmapInput_evt;
282     cl_event lastMarshalling_evt;
283 
284     cl_event**  run_start_evt; //NUM_QUEUES for each round
285     cl_event**  run_done_evt; //NUM_QUEUES for each round
286 
287     ErlNifEnv* env;
288     ErlNifMutex* env_mtx;
289 
290     uint pInputC;
291     cl_command_queue cmdQs[NUM_QUEUES];
292     uint cmdQsC;
293     cl_kernel* kernels;
294     cl_mem** inputBuffersD;
295     cl_mem outputBufferD;
296 
297 
298     //Shared InputH/outputH. set by load, used by unload in marshalling.
299     //Unload never writes a segment that Load hasn't copied yet on the device;
300     //ensured by stage sync: Load < run < unload
301     double* pMappedOutputBufferH;
302     cl_mem* inputBuffersH;
303     cl_mem outputBufferH;
304 
305     ERL_NIF_TERM* currList;
306 
```

```
307
308     Map_fun_counters* counters;
309
310     uint numQueues;
311     uint numRounds;
312
313     uint uiNumElem;
314     uint uiNumElemSegment;
315     size_t szSegment;
316
317     uint numSegments;
318
319     ERL_NIF_TERM* outputTerms;
320
321
322     explicit MapLL_thread_params(
323
324         cl_event _mapOutput_evt,
325         cl_event _unmapInput_evt,
326         cl_event _lastMarshalling_evt,
327         cl_event**  _run_start_evt,
328         cl_event**  _run_done_evt,
329
330         ErlNifEnv* _env,
331         ErlNifMutex* _env_mtx,
332         uint _pInputC,
333         cl_command_queue* _cmdQs,
334         cl_kernel* _kernels,
335         cl_mem** _inputBuffersD,
336         cl_mem _outputBufferD,
337
338         cl_mem* _inputBuffersH,
339         cl_mem _outputBufferH,
340
341         ERL_NIF_TERM* _currList,
342
343         Map_fun_counters* _counters,
344
345         uint _numRounds,
346         uint _numQueues,
347
348         uint _uiNumElem,
349         uint _uiNumElemSegment,
350         size_t _szSegment,
351
352         ERL_NIF_TERM* _outputTerms
353     )
```

```
354     {
355
356         errorSignal = false;
357         requestedExit = false;
358
359         errorTerm = 0;
360
361         mapOutput_evt = _mapOutput_evt;
362         unmapInput_evt = _unmapInput_evt;
363
364         lastMarshalling_evt = _lastMarshalling_evt;
365
366         run_start_evt = _run_start_evt;
367         run_done_evt = _run_done_evt;
368
369         env = _env;
370         env_mtx = _env_mtx;
371
372         pInputC = _pInputC;
373
374         for(int i = 0; i < NUM_QUEUES; i++)
375             cmdQs[i] = _cmdQs[i];
376
377
378         kernels = _kernels;
379         inputBuffersD = _inputBuffersD;
380         outputBufferD = _outputBufferD;
381
382
383         inputBuffersH = _inputBuffersH;
384         outputBufferH = _outputBufferH;
385
386         currList = _currList;
387
388         counters = _counters;
389
390         numRounds = _numRounds;
391         numQueues = _numQueues;
392
393         uiNumElem = _uiNumElem;
394         uiNumElemSegment = _uiNumElemSegment;
395         szSegment = _szSegment;
396
397         outputTerms = _outputTerms;
398
399         numSegments = uiNumElem/uiNumElemSegment;
400
```

```
401
402     }
403
404
405     void signalError() {
406
407 #ifdef DEBUG
408         cerr << "DEBUG: MapLL - signalError().\n";
409 #endif
410
411         errorSignal = true;
412     }
413
414     void requestExit() {
415         requestedExit = true;
416     }
417
418 };
419
420 void destroyMutexes(ErlNifMutex* mtxs[], uint len) {
421
422     for (uint i = 0; i < len; ++i) {
423         enif_mutex_destroy(mtxs[i]);
424         mtxs[i] = NULL;
425     }
426 }
427
428
429 //struct conatining unmashalling threads' shared data
430 struct MapLL_loadStage_unmarshalling_thread_params {
431
432     Barrier* barrier;
433     MapLL_thread_params* map_params;
434
435     uint* segmentOffset;
436     double** pMappedInputBuffersH;
437
438     uint* idx;
439
440     cl_command_queue curr_cmdQ;
441
442     bool* seqMode;
443
444     cl_event** round_start;
445
446     bool errorSignal;
447     bool* global_errorSignal;
```

```
448     ERL_NIF_TERM errorTerm;
449 };
450
451 static void* mapLL_loadStage_unmarshalling_Thread(void* obj) {
452
453
454 #ifdef DEBUG
455     char msg[128];
456 #endif
457
458     std::pair<uint, MapLL_loadStage_unmarshalling_thread_params*> *id_load_params =
459     (std::pair<uint, MapLL_loadStage_unmarshalling_thread_params*>*) obj;
460
461     uint& i_input = id_load_params->first; //thread's ID
462     MapLL_loadStage_unmarshalling_thread_params*& load_params = id_load_params->
            second;
463
464
465     //local variables
466     MapLL_thread_params*& params = load_params->map_params;
467
468     Barrier*& barrier = load_params->barrier;
469
470     uint*& segmentOffset = (load_params->segmentOffset);
471
472     ErlNifEnv*& env = params->env;
473     ErlNifMutex*& env_mtx = params->env_mtx;
474     Map_fun_counters* &counters = params->counters;
475
476     cl_command_queue& curr_cmdQ = load_params->curr_cmdQ;
477     bool*& seqMode = (load_params->seqMode);
478     cl_event**& round_start_evts = (load_params->round_start);
479
480     uint*& idx = (load_params->idx);
481
482
483
484 #ifdef DEBUG
485     sprintf(msg, "DEBUG: Load[%u] - starting\n", i_input); cerr << msg;
486 #endif
487
488     //unmarshalling:  L -> H
489     //copy data:  H -> D
490     for (uint i_round = 0 ; i_round < params->numRounds; ++i_round) {
491
492
493         for(uint i_queue = 0 ; i_queue <  params->numQueues; i_queue++) {
```

```
494
495 #ifdef DEBUG
496             sprintf(msg, "DEBUG: Load[%u] - awaiting signal [%u][%u] \n", i_input,
                    i_round,i_queue); cerr << msg;
497 #endif
498
499             //wait for signal from load master thread
500             clWaitForEvents(1, &(round_start_evts[i_round][i_queue]));
501
502 #ifdef DEBUG
503             sprintf(msg, "DEBUG: Load[%u] - starting [%u][%u] \n", i_input, i_round,
                    i_queue); cerr << msg;
504 #endif
505
506
507             //some other thread (unmarshalling or one of map's threads) has signaled
                     an error, exit
508             if(load_params->errorSignal || *(load_params->global_errorSignal)) {
509
510 #ifdef DEBUG
511                 sprintf(msg, "DEBUG: Load[%u] - Someone signaled an error, exit.\n",
                        i_input); cerr << msg;
512 #endif
513                 goto cleanup;
514             }
515
516
517 #ifdef TIME
518             GET_TIME((counters->round[i_round]->load_input_start[i_queue][i_input]))
                    ;
519 #endif
520
521             //************list -> H ************
522
523             double* pMappedInputBufferSegmentH = load_params->pMappedInputBuffersH[
                    i_input] + (*segmentOffset);
524
525             // copy current input list segment into the corresponding part of
                    bufferH
526             uint numUnmarshElems=
527             list_to_double_arrayN(
528                                     params->env,
529                                     params->currList[i_input],
530                                     pMappedInputBufferSegmentH, params->
                                         uiNumElemSegment,
531                                     &(params->currList[i_input])
532                                     );
```

```
533             //currList is updated to the first element of the next segment (of the
                    list)
534
535             if(numUnmarshElems != params->uiNumElemSegment)  {//some problem
                    happened during unmarshalling
536 #ifdef DEBUG
537                 sprintf(msg, "DEBUG: Load[%u] - [%u][%u] - Error unmarshalling\n",
                        i_input, i_round, i_queue); cerr << msg;
538 #endif
539
540                 load_params->errorSignal = true;
541                 load_params->errorTerm = sync_make_error(env, params->env_mtx, ATOM(
                        unmarshalling_error));
542
543                 return NULL;
544             }
545
546             //print_buffer_debug("inputBuffersH", inputBuffersH[i_input], oneIdx *
                    params->szSegment, params->szSegment); // DEBUG
547 #ifdef DEBUG
548             sprintf(msg, "DEBUG: Load[%u] - [%u][%u] - L -> H\n", i_input,  i_round,
                     i_queue );
549             cerr << msg;
550 #endif
551
552
553 #ifdef TIME
554             GET_TIME((counters->round[i_round]->load_unmarsh_end[i_queue][i_input]))
                    ;
555 #endif
556
557
558 #ifdef SEQ
559             *seqMode = true;
560 #endif
561
562             cl_event copy_Evt;
563             //************  H -> D ************
564             THREAD_CHK_SUCCESS_CLEANUP_GOTO(
565                 clEnqueueWriteBuffer(curr_cmdQ, params->inputBuffersD[i_input][*idx
                        ], CL_FALSE, 0, params->szSegment,
566                 (void*)pMappedInputBufferSegmentH, 0, NULL, *seqMode ? &copy_Evt :
                        NULL
567                 ); ,
568                 load_params->errorTerm,
569                 {
```

```
570                     sprintf(msg, "DEBUG: Load[%u] - [%u][%u] - ERROR H -> D \n",
                            i_input, i_round, i_queue); cerr << msg;
571                 load_params->errorSignal = true;
572                 }
573             )
574             clFlush(params->cmdQs[i_queue]);
575
576
577 #ifdef DEBUG
578             sprintf(msg, "DEBUG: Load[%u] - [%u][%u] - H -> D\n", i_input, i_round,
                    i_queue);
579             cerr << msg;
580 #endif
581
582 #ifdef SEQ
583             clWaitForEvents(1, &copy_Evt);
584             clReleaseEvent(copy_Evt);
585 #endif
586
587 #ifdef TIME
588             GET_TIME((counters->round[i_round]->load_copyHD_end[i_queue][i_input]));
589 #endif
590
591 #ifdef DEBUG
592             sprintf(msg, "DEBUG: Load[%u] - WAITING at barrier [%u][%u]\n", i_input,
                     i_queue, i_round); cerr << msg;
593 #endif
594             //synch with others unmarshalling threads
595             barrier->await();
596
597 #ifdef DEBUG
598             sprintf(msg, "DEBUG: Load[%u] - CROSSED barrier [%u][%u]\n", i_input,
                    i_queue, i_round); cerr << msg;
599 #endif
600
601         }
602     }
603
604 #ifdef DEBUG
605     sprintf(msg, "DEBUG: Load[%u] - DONE\n", i_input); cerr << msg;
606 #endif
607
608 cleanup:
609
610     //in case of error must clear the skipped barrier, so to avoid deadlocking other
          load threads
611     if(load_params->errorSignal || *(load_params->global_errorSignal)) {
```

```
612 #ifdef DEBUG
613         sprintf(msg, "DEBUG: Load[%u] - CLEARING skipped barrier\n", i_input); cerr
                << msg;
614 #endif
615         barrier->await();
616 
617     }
618 
619     return NULL;
620 
621 }
622 
623 
624 
625 static void* mapLL_loadStage_Thread(void* obj) {
626 
627 #ifdef DEBUG
628     char msg[256];
629 #endif
630 
631 #ifdef DEBUG
632     cerr << "DEBUG: Load - Starting.\n";
633 #endif
634 
635 
636     MapLL_thread_params* params = (MapLL_thread_params *) obj;
637 
638     //parametri
639 
640     //env, env_mtx needed by CLEANUP macros
641     ErlNifEnv* env = params->env;
642     ErlNifMutex* env_mtx = params->env_mtx;
643 
644     Map_fun_counters*& counters = params->counters;
645 
646     //local variables
647     size_t szInput_local = (params->szSegment / params->uiNumElemSegment) * params->
            uiNumElem;
648 
649     cl_int ciErrNum;
650 
651     double* pMappedInputBuffersH[params->pInputC];//pointers mapped into input
            buffers
652     bool inputBufferIsMapped = false;
653 
654     bool seqMode = false;//is sequential execution activated?
655 
```

```
656     //initilize synchronization barrier for coordinating #pInputC load threads +
            master load thread
657     Barrier round_end_barrier(params->pInputC + 1);
658
659
660     //check if load stage is configured (in mapLL)
661     bool unload_stage_isPresent = params->outputTerms != NULL ? true : false;
662
663
664 #ifdef TIME
665     GET_TIME(counters->mapInput_start);
666 #endif
667
668     //map InputBuffer
669     for(uint i_list = 0; i_list < params->pInputC; ++i_list) {
670
671         pMappedInputBuffersH[i_list] = (cl_double*)
672         clEnqueueMapBuffer(params->cmdQs[0], params->inputBuffersH[i_list], CL_TRUE,
                CL_MAP_WRITE, 0, szInput_local, 0, NULL, NULL, &ciErrNum);
673
674         THREAD_CHK_SUCCESS_CLEANUP_GOTO(
675             ciErrNum; ,
676             params->errorTerm,
677             {
678             cerr << "DEBUG: Load - clEnqueueMapBuffer\n";
679             params->signalError();
680             }
681         )
682 #ifdef TIME
683         GET_TIME(counters->mapInput_end[i_list]);
684 #endif
685     }
686
687     inputBufferIsMapped = true;
688
689     //set output buffer's pointer to mapped buffer memory, since inputH[0] is
            outputH
690     params->pMappedOutputBufferH = pMappedInputBuffersH[0];
691
692     //get OpenCL context needed by clCreateUserEvent
693     cl_context context;
694     clGetCommandQueueInfo(params->cmdQs[0], CL_QUEUE_CONTEXT, sizeof(cl_context), &
            context, NULL);
695
696     cl_event* round_start_evts[params->numRounds];
697
698     for (uint i_round = 0 ; i_round < params->numRounds; i_round++) {
```

```
699
700         round_start_evts[i_round] = new cl_event[params->numQueues];
701
702         for(uint i_queue = 0 ; i_queue <  params->numQueues; i_queue++)
703             round_start_evts[i_round][i_queue] = clCreateUserEvent(context, NULL);
704     }
705
706
707
708     MapLL_loadStage_unmarshalling_thread_params unmarsh_params;
709
710     //initialize params' loop-independent variables
711     unmarsh_params.barrier = &round_end_barrier;
712     unmarsh_params.map_params = params;
713     unmarsh_params.pMappedInputBuffersH = pMappedInputBuffersH;
714
715     unmarsh_params.seqMode = &seqMode;
716     unmarsh_params.round_start = round_start_evts;
717
718     unmarsh_params.errorSignal = false;//unmarshalling threads signal (local)
719     unmarsh_params.global_errorSignal = &params->errorSignal; //map threads signal (
        global)
720
721     //create unmarshalling threads
722     ErlNifTid unmarshalling_threads[params->pInputC];
723
724     std::pair<uint, MapLL_loadStage_unmarshalling_thread_params*> *id_params[params
        ->pInputC];//(Thread-ID, thread params)
725
726     for(uint i_input = 0; i_input < params->pInputC; i_input++) {
727
728         id_params[i_input] = new std::pair<uint,
            MapLL_loadStage_unmarshalling_thread_params*>(i_input, &unmarsh_params);
729
730         enif_thread_create((char*)"Unmarshalling", &unmarshalling_threads[i_input],
            mapLL_loadStage_unmarshalling_Thread, (void*) id_params[i_input], NULL);
731     }
732
733
734     /******LOAD LOOP*****/
735     uint zeroIdx;//first segment index (the one that will be computed by the first
        queue)
736     uint i_round;
737     for (i_round = 0 ; i_round < params->numRounds; ++i_round) {
738
739         zeroIdx = (params->numQueues * i_round);
740
```

```
741         uint idx;
742         uint segmentOffset;
743 
744 
745 #ifdef DEBUG
746         sprintf(msg, "DEBUG: Load - Starting Round[%d]\n", i_round);
747         cerr << msg;
748 #endif
749 
750         for(uint i_queue = 0 ; i_queue <  params->numQueues; i_queue++) {
751 
752 #ifdef TIME
753             GET_TIME((params->counters->round[i_round]->start[i_queue]));
754 
755             params->counters->round[i_round]->load_start[i_queue] =
756             params->counters->round[i_round]->start[i_queue];
757 
758 #endif
759 
760             idx = zeroIdx + i_queue;
761 
762             segmentOffset = (idx * params->uiNumElemSegment);
763 
764             cl_command_queue& curr_cmdQ = params->cmdQs[i_queue];
765 
766             cl_event& curr_run_start_evt = params->run_start_evt[i_queue][i_round];
767 
768 
769             if(i_round > 0) { //from the first round onwards
770                 uint round_to_wait = i_round - 1;
771                 cl_event& curr_run_done_evt = params->run_done_evt[i_queue][
                        round_to_wait];
772 
773 #ifdef DEBUG
774                 sprintf(msg, "DEBUG: Load - %d - Waiting for unload[%d][%d] to start
                        .\n", i_queue, i_queue, round_to_wait);
775                 cerr << msg;
776 #endif
777                 //WAIT for RUN stage, so to synch with unload stage
778                 clWaitForEvents(1, &(curr_run_done_evt));
779 
780             }
781 
782 
783             if(params->errorSignal) { //someone has signaled an error, cleanup and
                    exit
784 #ifdef DEBUG
```

```
785                 sprintf(msg, "DEBUG: Load - %d - Someone has signaled an error,
                        cleanup and exit\n", i_queue);
786                 cerr << msg;
787 #endif
788                 goto cleanup;
789             }
790
791             /*
792              * list -> H
793              * H -> D
794              */
795
796             //update unmarshalling threads' loop-dependent variables
797             unmarsh_params.curr_cmdQ = curr_cmdQ;
798             unmarsh_params.idx = &idx;
799             unmarsh_params.segmentOffset = &segmentOffset;
800
801
802 #ifdef DEBUG
803             sprintf(msg, "DEBUG: Load - signaling unmarshalling threads [%d][%d]\n",
                    i_round,i_queue); cerr << msg;
804 #endif
805             //signal unmarshalling threads
806             clSetUserEventStatus(round_start_evts[i_round][i_queue], CL_SUCCESS);
807
808
809 #ifdef DEBUG
810             sprintf(msg, "DEBUG: Load - WAITING at barrier [%d][%d]\n",i_round,
                    i_queue); cerr << msg;
811 #endif
812             //await unmarshalling completion
813             round_end_barrier.await();
814
815 #ifdef DEBUG
816             sprintf(msg, "DEBUG: Load - barrier CROSSED [%d][%d]\n",i_round,i_queue)
                    ; cerr << msg;
817 #endif
818
819             //check if everything went fine, otherwise set error term (i.e.
                    propagate the error) and cleanup
820             if(unmarsh_params.errorSignal){
821
822 #ifdef DEBUG
823                 sprintf(msg, "DEBUG: Load - some unmarshalling thread signaled an
                        error, cleanup.\n"); cerr << msg;
824 #endif
825
```

```
826                 params->errorTerm = unmarsh_params.errorTerm;
827                 params->signalError();
828 
829                 goto cleanup;
830             }
831 
832             if(params->errorTerm){
833 
834                 goto cleanup;
835             }
836 
837             //******Segment is now loaded on device******
838 
839 #ifdef TIME
840             GET_TIME((counters->round[i_round]->load_end[i_queue]));
841 #endif
842 
843             //signal run stage, so it can process the segment i just loaded
844             clSetUserEventStatus(curr_run_start_evt, CL_SUCCESS);
845 
846 
847         }//queue loop end
848 
849     }//round loop end
850 
851 #ifdef DEBUG
852     sprintf(msg, "DEBUG: Load - WAITING for unload, must UnmapInput\n"); cerr << msg
        ;
853 #endif
854 
855 
856     if(unload_stage_isPresent) {
857         //overlap unmapping with last marshalling (which is done by unload thread)
858         clWaitForEvents(1, &params->unmapInput_evt);
859     }
860 
861 
862 #ifdef TIME
863     GET_TIME((counters->unmapInput_start));
864 #endif
865 
866     if(inputBufferIsMapped) {
867 
868 #ifdef DEBUG
869         sprintf(msg, "DEBUG: Load - UNMAPPING Input\n"); cerr << msg;
870 #endif
871 
```

```
872         //unmap input buffers
873         for(uint i_list = 1/*0*/; i_list < params->pInputC; ++i_list)
874             // Starts from 1; 0 is released by unload since it's used also as output
875             clEnqueueUnmapMemObject(params->cmdQs[0], params->inputBuffersH[i_list],
                    pMappedInputBuffersH[i_list], 0, NULL, NULL);
876
877 #ifdef TIME
878         GET_TIME((counters->unmapInput_end));
879 #endif
880     }
881     inputBufferIsMapped = false;
882
883
884
885 #ifdef DEBUG
886     sprintf(msg, "DEBUG: Load - CLEANING UP\n"); cerr << msg;
887 #endif
888
889
890
891
892 cleanup:
893
894     //in case of error
895     if(params->errorSignal || unmarsh_params.errorSignal) {//release every lock
        before exiting to avoid deadlocks
896
897 #ifdef DEBUG
898         sprintf(msg, "DEBUG: Load - Error detected, set all events\n"); cerr << msg;
899 #endif
900
901         for (uint i_round = 0 ; i_round < params->numRounds; ++i_round)
902             for (uint i_queue = 0 ; i_queue < params->numQueues; ++i_queue) {
903
904                 clSetUserEventStatus(params->run_start_evt[i_queue][i_round],
                        CL_SUCCESS);
905
906                 clSetUserEventStatus(round_start_evts[i_round][i_queue], CL_SUCCESS)
                        ;
907             }
908
909     }
910
911 #ifdef DEBUG
912     sprintf(msg, "DEBUG: Load - joining unmarshalling threads\n"); cerr << msg;
913 #endif
914     //join unmarshalling threads
```

```
915     for(uint i_input = 0; i_input < params->pInputC; i_input++)
916         enif_thread_join(unmarshalling_threads[i_input], NULL);
917
918
919     if(inputBufferIsMapped) {
920         //unmap input buffers
921         for(uint i_list = 1/*0*/; i_list < params->pInputC; ++i_list)// Start from
                1. 0 released by unload
922             clEnqueueUnmapMemObject(params->cmdQs[0], params->inputBuffersH[i_list],
                    pMappedInputBuffersH[i_list], 0, NULL, NULL);
923     }
924
925
926 #ifdef DEBUG
927     sprintf(msg, "DEBUG: Load - RELEASING Input buffers [1,N_Input]\n"); cerr << msg
            ;
928 #endif
929     //release inputBuffersH[] szInput
930     for (uint i_list = 0; i_list < params->pInputC; ++i_list) //starts from 0 (
            unlike unmapping) because inputBuffersH[0] has been retained upon creation
            to allow it (by Master)
931         clReleaseMemObject(params->inputBuffersH[i_list]);
932
933 #ifdef TIME
934     GET_TIME((counters->releaseInBuffersH_T));
935 #endif
936
937
938 #ifdef DEBUG
939     sprintf(msg, "DEBUG: Load - DELETING id-params pairs\n"); cerr << msg;
940 #endif
941
942     //delete id-params pairs
943     for (uint i_input = 0; i_input < params->pInputC; ++i_input)
944         delete id_params[i_input];
945
946
947 #ifdef DEBUG
948     sprintf(msg, "DEBUG: Load - RELEASING round_start_events\n"); cerr << msg;
949 #endif
950
951     //release events
952     for (uint i_round = 0 ; i_round < params->numRounds; i_round++) {
953
954         for(uint i_queue = 0 ; i_queue <  params->numQueues; i_queue++)
955             clReleaseEvent(round_start_evts[i_round][i_queue]);
956
```

```
957         delete[] round_start_evts[i_round];
958     }
959
960
961
962 #ifdef DEBUG
963     sprintf(msg, "DEBUG: Load - DONE\n");
964     cerr << msg;
965 #endif
966
967     return NULL;
968 }
969
970
971
972
973 static void* mapLL_runStage_Thread(void* obj) {
974 #ifdef DEBUG
975     char msg[256];
976 #endif
977
978     MapLL_thread_params* params = (MapLL_thread_params *) obj;
979
980     //local
981     //  bool seqMode = false;
982
983     //needed by CLEANUP macros
984     ErlNifEnv*& env = params->env;
985     ErlNifMutex*& env_mtx = params->env_mtx;
986
987     Map_fun_counters*& counters = params->counters;
988
989     size_t szGlobalWorkSize =  params->uiNumElemSegment;
990
991     bool isLastStage = params->outputTerms == NULL ? true : false;//Am I the last
        stage of the pipeline?
992
993     uint zeroIdx = 0;
994     for (uint i_round = 0 ; i_round < params->numRounds; ++i_round) {
995
996         zeroIdx = (params->numQueues * i_round);
997
998
999 #ifdef DEBUG
1000         sprintf(msg, "DEBUG: Run - Starting round[%d]\n", i_round);
1001         cerr << msg;
1002 #endif
```

```
1003
1004         uint idx = 0;
1005         for (uint i_queue = 0 ; i_queue < params->numQueues; ++i_queue) {
1006
1007
1008 #ifdef DEBUG
1009             sprintf(msg, "DEBUG: Run - %d - waiting to start\n", i_queue);
1010             cerr << msg;
1011 #endif
1012
1013             idx = zeroIdx + i_queue;
1014
1015             //current loop values
1016             cl_kernel& curr_kernel =  params->kernels[i_queue];
1017             cl_command_queue& curr_cmdQ = params->cmdQs[i_queue];
1018
1019             cl_event& curr_run_start_evt = params->run_start_evt[i_queue][i_round];
1020             cl_event& curr_run_done_evt = params->run_done_evt[i_queue][i_round];
1021
1022             uint segmentOffset = idx * params->uiNumElemSegment;
1023
1024
1025             //wait for load stage
1026             clWaitForEvents(1, &(curr_run_start_evt));
1027
1028
1029             if(params->errorSignal) { //Someone signaled an error, exit.
1030 #ifdef DEBUG
1031                 sprintf(msg, "DEBUG: Run - %d - Someone signaled an error, exit.\n",
                        i_queue);
1032                 cerr << msg;
1033 #endif
1034                 goto cleanup;
1035             }
1036
1037
1038             //*********set map kernels parameters**********************
1039
1040
1041             uint i_input = 0;
1042
1043             uint numKerParams = params->pInputC + 3;
1044
1045             cl_int ciErrNum[numKerParams];
1046
1047
1048
```

```
1049            for(i_input=0; i_input < params->pInputC; i_input++)
1050                ciErrNum[i_input] = clSetKernelArg(curr_kernel, i_input, sizeof(
                        cl_mem), (void*) &(params->inputBuffersD[i_input][idx]));
1051
1052            i_input--;
1053
1054            ciErrNum[i_input+1] = clSetKernelArg(curr_kernel, i_input+1, sizeof(
                    cl_mem), (void*) &(params->outputBufferD));//round independent
1055            ciErrNum[i_input+2] = clSetKernelArg(curr_kernel, i_input+2, sizeof(
                    unsigned int), (void*) &(segmentOffset));
1056
1057            ciErrNum[i_input+3] = clSetKernelArg(curr_kernel, i_input+3, sizeof(
                    unsigned int), (void*) &(params->uiNumElemSegment));//round
                    independent
1058
1059
1060            for(uint i = 0; i < numKerParams; i++) {
1061                if(ciErrNum[i] != CL_SUCCESS) {
1062 #ifdef DEBUG
1063                    sprintf(msg, "DEBUG: Run - %d - Error SetKernel #%d\n", i_queue,
                            i); cerr << msg;
1064 #endif
1065                    params->signalError();
1066                    params->errorTerm = sync_make_error_cl(env, env_mtx, ciErrNum[i
                            ]);
1067
1068                    goto cleanup;
1069                }
1070            }
1071
1072
1073 #ifdef DEBUG
1074            sprintf(msg, "DEBUG: Run - %d - Kernel Set\n", i_queue);
1075            cerr << msg;
1076 #endif
1077
1078
1079 #ifdef TIME
1080            GET_TIME((counters->round[i_round]->run_start[i_queue]));
1081 #endif
1082
1083
1084 #ifdef SEQ
1085            seqMode = true;
1086 #endif
1087
1088            //************ Run ************
```

```
1089             cl_event exec_Evt;
1090             THREAD_CHK_SUCCESS_CLEANUP_GOTO(
1091               clEnqueueNDRangeKernel(curr_cmdQ, curr_kernel, 1, NULL, &
                       szGlobalWorkSize, NULL/*&szLocalWorkSize*/, 0, NULL, &exec_Evt); ,
1092               params->errorTerm,
1093               {
1094               sprintf(msg, "DEBUG: Run - %d - Error NDRangeKernel\n", i_queue); cerr
                       << msg;
1095               params->signalError();
1096               }
1097             )
1098             clFlush(curr_cmdQ);
1099
1100
1101 #ifdef DEBUG
1102             sprintf(msg, "DEBUG: Run - %d - run\n", i_queue);
1103             cerr << msg;
1104 #endif
1105
1106
1107             clWaitForEvents(1, &exec_Evt);
1108             clReleaseEvent(exec_Evt);
1109
1110
1111
1112 #ifdef TIME
1113             GET_TIME((counters->round[i_round]->run_end[i_queue]));
1114             if(isLastStage)
1115                 counters->round[i_round]->end[i_queue] = counters->round[i_round]->
                       run_end[i_queue];
1116 #endif
1117
1118             //     clWaitForEvents(1, &exec1_Evt);
1119
1120
1121
1122 #ifdef DEBUG
1123             sprintf(msg, "DEBUG: Run - %d - wake up unload\n", i_queue);
1124             cerr << msg;
1125 #endif
1126             //signal unload stage
1127             clSetUserEventStatus(curr_run_done_evt, CL_SUCCESS);
1128
1129         }//end queue
1130
1131     }//end round
1132
```

```
1133 cleanup:
1134
1135     //in case of error
1136     if(params->errorSignal)  {//to avoid deadlocks, release every lock before
             exiting
1137
1138 #ifdef DEBUG
1139         sprintf(msg, "DEBUG: Run - Error detected, set all events\n");
1140         cerr << msg;
1141 #endif
1142
1143         for (uint i_round = 0 ; i_round < params->numRounds; ++i_round)
1144             for (uint i_queue = 0 ; i_queue < params->numQueues; ++i_queue)
1145                 clSetUserEventStatus(params->run_done_evt[i_queue][i_round],
                        CL_SUCCESS);
1146     }
1147
1148 #ifdef DEBUG
1149     sprintf(msg, "DEBUG: Run - Done\n");
1150     cerr << msg;
1151 #endif
1152
1153     return NULL;
1154 }
1155
1156
1157 static ERL_NIF_TERM mapLD_Impl_3Thread(ErlNifEnv * env, kernel_sync* pKernel_s, void
        ** _pInputV, uint pInputC, void* _pOutputBuffer, uint listLength) {
1158
1159 #ifdef DEBUG
1160     cerr << "DEBUG: M - mapLD_Impl_3Thread" << endl;
1161 #endif
1162
1163     ERL_NIF_TERM toReturn;
1164
1165
1166     const uint NUM_SEGMENTS = NUM_SEGM;
1167
1168     const uint NUM_QUEUES_LOCAL = NUM_QUEUES;
1169
1170     const uint numRounds = NUM_SEGMENTS/NUM_QUEUES_LOCAL;
1171
1172
1173 #if defined(TIME) || defined(TIME_FUN)
1174
1175 #define DIFF_T0(COUNTER) nsec2usec(diff( &(counters->fun_start), &(COUNTER)))
1176
```

```
1177     Map_fun_counters* counters = new Map_fun_counters(numRounds, pInputC);
1178 #endif
1179
1180 #ifdef TIME
1181
1182     timespec createEventsT_start, createEventsT_end,
1183     createInputBuffersH[pInputC],
1184     createInputBuffersD[pInputC],
1185
1186     createOutputBufferD;
1187
1188     timespec joinT[2],//3
1189     //list_createdT_start, list_createdT_end,
1190     releaseEventsT;
1191
1192     //  clock_getres(CLOCK_MONOTONIC, &(counters->fun_start));
1193     //  cerr << "INFO : Time resolution is " << counters->fun_start.tv_nsec << "
            nsecs" << endl;
1194
1195 #endif
1196
1197 #if defined(TIME) || defined(TIME_FUN)
1198     GET_TIME(counters->fun_start);
1199 #endif
1200
1201
1202     std::vector<cl_event> events;
1203
1204     //cast input/output to type relevant here
1205     cl_mem outputBufferD = *((cl_mem*) _pOutputBuffer);
1206
1207     //pInputV is an array of ERL_NIF_TERM* representing, each one, an erlang list
1208     ERL_NIF_TERM** inputLists = (ERL_NIF_TERM**) _pInputV;
1209
1210     //input lists must have at least minListLen elements,
1211     //otherwise they can't be split in NUM_SEGMENTS segments
1212     uint listLen = listLength;
1213
1214
1215     uint minListLen = device_minBaseAddrAlignByte / sizeof(double);
1216     if(listLen/NUM_SEGMENTS < minListLen)
1217         return make_error(env, ATOM(skel_ocl_input_list_too_short));
1218
1219
1220     uint uiNumElem = listLen;
1221
1222     uint uiNumElemSegment = uiNumElem / NUM_SEGMENTS;
```

```
1223
1224     size_t szInput = uiNumElem * sizeof(double);
1225
1226     size_t szSegment = szInput / NUM_SEGMENTS;
1227
1228
1229 #ifdef DEBUG
1230     cerr << "DEBUG: M - mapLD_impl:" << "#Elem: "<< uiNumElem << " #ElemSegment: "<<
          uiNumElemSegment <<" szInput: "<< szInput << " szSegment: "<< szSegment <<
         endl;
1231 #endif
1232
1233     cl_int ciErrNum = 0;
1234
1235     //working queues
1236     cl_command_queue cmdQs[2] = {getCommandQueue(0), getCommandQueue(1)};
1237
1238     cl_context context = getContext();
1239
1240     //mutex protecting env from concurrent modifications (enif_make_xxx)
1241     ErlNifMutex* env_mtx = enif_mutex_create((char*)"env_mtx");
1242
1243
1244 #ifdef TIME
1245     GET_TIME(createEventsT_start);
1246 #endif
1247
1248     //********events setup [NUM_QUEUES][numRounds])
1249     cl_event* run_start_evt[NUM_QUEUES_LOCAL];
1250     cl_event* run_done_evt[NUM_QUEUES_LOCAL];
1251
1252     for (int i = 0; i < NUM_QUEUES_LOCAL; ++i) {
1253         run_start_evt[i] = new cl_event[numRounds];
1254         run_done_evt[i] = new cl_event[numRounds];
1255
1256         for (int i_round = 0; i_round < numRounds; ++i_round) {
1257             run_start_evt[i][i_round] = clCreateUserEvent(context, &ciErrNum);
1258             run_done_evt[i][i_round] = clCreateUserEvent(context, &ciErrNum);
1259
1260             events.push_back(run_start_evt[i][i_round]);
1261             events.push_back(run_done_evt[i][i_round]);
1262         }
1263     }
1264
1265 #ifdef TIME
1266     GET_TIME(createEventsT_end);
1267 #endif
```

```
1268
1269        //*********threads setup
1270        //Load, run. Each stage works on 2 command queues
1271        const uint NUM_STAGES = 2;
1272        ErlNifTid tid[NUM_STAGES];
1273
1274        ErlNifTid& loadStage = tid[0];
1275        ErlNifTid& runStage = tid[1];
1276
1277        //object holding everything needed by the threads
1278        MapLL_thread_params* conf = NULL;
1279
1280
1281        //***********for each input list, create:
1282        // one inputBufferH
1283        // NUM_SEGMENTS inputBufferD
1284
1285        cl_mem inputBuffersH[pInputC];
1286
1287        //[pInputC][NUM_SEGMENTS]
1288        cl_mem* inputBuffersD[pInputC];
1289        for (int i = 0; i < pInputC; ++i)
1290            inputBuffersD[i] = new cl_mem[NUM_SEGMENTS];
1291
1292        for(uint i_list = 0; i_list < pInputC; ++i_list) {
1293            // create one input host buffer of size szInput, released by Unload
1294            CHK_SUCCESS_CLEANUP_GOTO(
1295                                        createBuffer(szInput, CL_MEM_READ_WRITE |
                                                CL_MEM_ALLOC_HOST_PTR, &(inputBuffersH[i_list])
                                                ); ,
1296                                        ;
1297                                        )
1298
1299 #ifdef TIME
1300            GET_TIME(createInputBuffersH[i_list]);
1301 #endif
1302
1303
1304 #ifdef DEBUG
1305            cerr << "DEBUG: M - " << "inputBuffersH[" << i_list << "] CREATED: size: "
                    << szInput << endl;
1306 #endif
1307
1308
1309            //*********create NUM_SEGMENTS input device buffers having size szSegment
                    *********
1310            //released by Unload
```

```
1311         for (int i_segm = 0; i_segm < NUM_SEGMENTS; ++i_segm) {
1312
1313             CHK_SUCCESS_CLEANUP_GOTO(
1314                                     createBuffer(szSegment, CL_MEM_READ_ONLY, &(
                                            inputBuffersD[i_list][i_segm]) ); ,
1315                                     ;
1316                                     )
1317
1318
1319 #ifdef TIME
1320             GET_TIME(createInputBuffersD[i_list]);
1321 #endif
1322
1323 #ifdef DEBUG
1324             cerr << "DEBUG: M - " << "inputBuffersD["<<i_list<<"]["<<i_segm<<"]
                    CREATED: size: " << szSegment<< endl;
1325 #endif
1326         }
1327 #ifdef DEBUG
1328         cerr << "DEBUG: M - " << "inputBuffersD CREATED"<< endl;
1329 #endif
1330
1331     }
1332
1333
1334 #ifdef TIME
1335     GET_TIME(createOutputBufferD);
1336 #endif
1337
1338
1339     //Clone kernel, 'cause i need two of them, one per queue
1340     cl_kernel kernels[NUM_QUEUES_LOCAL];
1341
1342     kernels[0] = pKernel_s->kernel;
1343
1344     if(NUM_QUEUES_LOCAL == TWO)
1345         cloneKernel(pKernel_s->kernel, &kernels[1]);
1346
1347     if(NUM_QUEUES_LOCAL > MAX_QUEUES) {
1348         toReturn = make_error(env, ATOM(ocl_num_queue_not_supported));
1349         goto cleanup;
1350     }
1351
1352
1353
1354     //*************************Main LOOP**********************
1355
```

```
1356
1357     ERL_NIF_TERM currList[pInputC]; //iterator on the list, points to the head of
             the current segment of the list
1358     for (uint i = 0; i < pInputC; ++i)
1359         currList[i] = *(inputLists[i]);
1360
1361
1362     conf = new MapLL_thread_params (
1363                                     NULL,//mapOutput_evt,
1364                                     NULL,//unmapInput_evt,
1365                                     NULL,//lastMarshalling_evt,
1366                                     run_start_evt,
1367                                     run_done_evt,
1368                                     env, env_mtx,
1369                                     pInputC, cmdQs, kernels,
1370                                     inputBuffersD, outputBufferD,
1371                                     inputBuffersH,
1372                                     NULL, //outputBufferH,
1373                                     currList,
1374 #ifdef TIME
1375                                     counters,
1376 #else
1377                                     NULL,
1378 #endif
1379                                     numRounds,
1380                                     NUM_QUEUES_LOCAL,
1381                                     uiNumElem, uiNumElemSegment, szSegment,
1382                                     NULL//outputTerms
1383                                     );
1384
1385
1386 #ifdef DEBUG
1387     cerr << "DEBUG: M - Starting threads." << endl;
1388 #endif
1389
1390     enif_thread_create((char*) "mapLL loadStage", &loadStage, mapLL_loadStage_Thread
             , conf, NULL);
1391
1392     enif_thread_create((char*) "mapLL runStage", &runStage, mapLL_runStage_Thread,
             conf, NULL);
1393
1394
1395 #ifdef TIME
1396     GET_TIME((counters->fun_prologue));
1397 #endif
1398
1399
```

```
1400     if(conf->errorSignal) {
1401         //something's gone wrong, cleanup and return the error term (the last one
                 generated)
1402 #ifdef DEBUG
1403         cerr << "DEBUG: M - Got an error: cleanup and exit"<< endl;
1404 #endif
1405 
1406         toReturn = conf->errorTerm;
1407     }
1408 
1409     //wait for termination
1410     for(uint i = 0; i < NUM_STAGES; ++i) {
1411 
1412         enif_thread_join(tid[i], NULL);
1413 
1414 #ifdef TIME
1415         GET_TIME((joinT[i]));
1416 #endif
1417 
1418 #ifdef DEBUG
1419         char msg[64];
1420         sprintf(msg, "DEBUG: M - Thread %d joined\n",i);
1421         cerr << msg;
1422 #endif
1423     }
1424 
1425     //check again after joining, shouldn't be useful since list creation starts
             after the last marshalling, just before outputBuffer's unmapping,
1426     //at the very end of the computation when nobody can signal an error anymore.
1427     if(conf->errorSignal) {
1428         //something's gone wrong, cleanup and return the error term (the last one
                 generated)
1429 #ifdef DEBUG
1430         cerr << "DEBUG: M - Got an error: cleanup and exit"<< endl;
1431 #endif
1432         toReturn = conf->errorTerm;
1433     }
1434 
1435     //******************CLEANUP label******************
1436 cleanup:
1437 
1438 
1439     //******************EVENTS******************
1440     if(!events.empty())
1441         releaseEvents(events);
1442 
1443     for(uint i = 0; i < NUM_QUEUES_LOCAL; i++){
```

```
1444         delete [] run_start_evt[i];
1445         delete [] run_done_evt[i];
1446     }
1447 #ifdef TIME
1448     GET_TIME((releaseEventsT));
1449 #endif
1450 
1451     //******************KERNELS******************
1452     if(NUM_QUEUES_LOCAL == 2){
1453         if(kernels[1])
1454             clReleaseKernel(kernels[1]); //2 usecs
1455         kernels[1] = NULL;
1456     }
1457 
1458     //******************ENV MUTEX******************
1459     if(env_mtx)
1460         destroyMutexes(&env_mtx, 1);//1 usec
1461 
1462     //******************CONF******************
1463     if(conf)
1464         delete conf;
1465 
1466 
1467 #ifdef DEBUG
1468     cerr << "DEBUG: M - Cleanup done. Return"<< endl;
1469 #endif
1470 
1471 
1472 #if defined(TIME) || defined(TIME_FUN)
1473     GET_TIME((counters->fun_end));
1474 #endif
1475 
1476 
1477 
1478     //************************RUN Statistics********************
1479 
1480 
1481 #if defined(TIME) || defined(TIME_FUN)
1482 
1483     cerr << endl <<
1484     "------------------------------------ map"<< pInputC <<"LD: (usec)
            ------------------------------------" <<
1485     endl <<
1486     "Processing " << pInputC << " X " << uiNumElem <<" elements in "<< NUM_SEGMENTS
            <<" segments (" << uiNumElemSegment <<" per segment)."<<
1487     endl <<
1488     "Total run time: " << DIFF_T0(counters->fun_end) <<
```

```
1489     endl;
1490
1491 #endif
1492
1493 #ifdef TIME
1494
1495     long prologueTime = DIFF_T0(counters-> fun_prologue);
1496
1497
1498     cerr << endl <<
1499     "Function prologue time: " <<  prologueTime << "\t\t\t\t\tT: "<< prologueTime <<
1500     endl;
1501
1502     char str[128];
1503     sprintf(str, "\tcreateEvents: %lu \t\t\t\t\tT: %lu\n",  DIFF(createEventsT_start
         , createEventsT_end), DIFF_T0( createEventsT_end)); cerr << str;
1504
1505     for (uint i_input = 0; i_input < pInputC; i_input++) {
1506         sprintf(str, "\tcreateInputBuffersH[%d]: \t\t\t\tT: %lu\n", i_input, DIFF_T0
             ( createInputBuffersH[i_input])); cerr << str;
1507
1508         sprintf(str,
1509                 "Load - mapInputBuffers[%d]: %lu \t\t\t\tT: %lu\n", i_input,
1510                 i_input == 0 ?
1511                 DIFF( counters->mapInput_start, counters->mapInput_end[i_input]) :
                        DIFF( counters->mapInput_end[i_input-1], counters->mapInput_end[
                        i_input]),
1512                 DIFF_T0( counters->mapInput_end[i_input])
1513                 );
1514         cerr << str;
1515
1516
1517         sprintf(str, "\tcreateInputBuffersD[%d]: \t\t\t\tT: %lu\n", i_input, DIFF_T0
             ( createInputBuffersD[i_input])); cerr << str;
1518     }
1519
1520     uint kernelTotal = 0, inputTotal = 0, outputTotal = 0, marshTotal = 0,
         unmarshTotal = 0;
1521     uint i_round;
1522     //round statistics
1523     for (i_round = 0; i_round < numRounds; i_round++) {
1524         uint tKernel[2] = {0,0};
1525         uint tInput = 0, tOutput = 0;
1526         uint tStartRound[2];
1527
1528         cerr <<
```

```
1529        "\nRound["<< i_round << "] total time: " << DIFF(( counters->round[i_round
                ]->start[0]), ( counters->round[i_round]->end[NUM_QUEUES_LOCAL-1]))<<
1530        endl;
1531
1532
1533        for (int i_queue = 0; i_queue < NUM_QUEUES_LOCAL; i_queue++) {
1534
1535            inputTotal +=
1536            tInput = DIFF(( counters->round[i_round]->load_start[i_queue]), (
                    counters->round[i_round]->load_end[i_queue]));
1537
1538
1539            tStartRound[i_queue] = DIFF_T0(counters->round[i_round]-> start[i_queue
                    ]);
1540            cerr <<
1541            "\n\t"<< i_queue << " - R["<< i_round << "] Start round\t\t\t\t\tT: " <<
                     tStartRound[i_queue]<<
1542            endl <<
1543            "\t"<< i_queue<< " - R["<< i_round << "] Load total time: " << tInput <<
1544            endl;
1545
1546            for (uint i_input = 0; i_input < pInputC; i_input++) {
1547                long unmarsh = 0, t_Unmarsh;
1548                long cpDH_0 = 0, t_cpDH_0;
1549
1550                unmarshTotal +=
1551                unmarsh = DIFF((counters->round[i_round]->load_input_start[i_queue][
                        i_input]), (counters->round[i_round]->load_unmarsh_end[i_queue][
                        i_input]));
1552
1553                t_Unmarsh = DIFF_T0(counters->round[i_round]-> load_unmarsh_end[
                        i_queue][i_input]);
1554
1555
1556                cpDH_0 = DIFF((counters->round[i_round]->load_unmarsh_end[i_queue][
                        i_input]), (counters->round[i_round]->load_copyHD_end[i_queue][
                        i_input]));
1557                t_cpDH_0 = DIFF_T0(counters->round[i_round]-> load_copyHD_end[
                        i_queue][i_input]);
1558
1559                cerr <<
1560                "\t\t"<< i_queue << " - R["<< i_round << "] Input["<< i_input<<"]
                        unmarshall: " << unmarsh << "\t\tT: "<<  t_Unmarsh <<
1561                endl <<
1562                "\t\t"<< i_queue << " - R["<< i_round << "] Input["<< i_input<<"] H
                        -> D: " << cpDH_0 << "\t\t\tT: "<<  t_cpDH_0 <<
1563                endl;
```

```
1564            }
1565
1566
1567
1568            long t_Kernel = DIFF_T0(counters->round[i_round]-> run_end[i_queue]);
1569            kernelTotal +=
1570            tKernel[i_queue] = DIFF((counters->round[i_round]->run_start[i_queue]),
                    (counters->round[i_round]->run_end[i_queue]));
1571
1572            cerr <<
1573            "\t" << i_queue<< " - R["<< i_round << "] KERNEL computation time: " <<
                    tKernel[i_queue] << "\t\t\tT: "<<  t_Kernel <<
1574            endl;
1575            endl;
1576
1577        }// end queues
1578    }//end rounds
1579
1580    //FROM last load (lastRound, last_queue) TO unmapInput_end
1581    cerr << "Load - unmapInputBuffers: "<< DIFF(counters->unmapInput_start, counters
        ->unmapInput_end) <<
1582    "\t\t\t\t\tT: " << DIFF_T0( counters->unmapInput_end)
1583    << endl;
1584
1585    cerr << "Load - releaseInBuffersH: " << DIFF(counters->unmapInput_end, counters
        ->releaseInBuffersH_T) <<"\t\t\t\t\tT: " << DIFF_T0( counters->
        releaseInBuffersH_T) << "\n";
1586
1587
1588    for(int i = 0; i < NUM_STAGES; i++)
1589        cerr << "M - join "<< i << "\t\t\t\t\t\tT: " << DIFF_T0( joinT[i]) << "\n"
                ;
1590
1591
1592    cerr << "M - releaseEvents: "<< DIFF(joinT[NUM_STAGES-1], releaseEventsT ) << "\
        t\t\t\t\t\tT: " << DIFF_T0( releaseEventsT) << "\n";
1593
1594    cerr << endl <<
1595    "KERNEL total time: " << kernelTotal << ". avg on "<< numRounds <<" rounds: " <<
         kernelTotal/ NUM_SEGMENTS <<
1596    endl <<
1597    "Load time: " << inputTotal << ". avg: " << inputTotal / NUM_SEGMENTS <<
1598    endl <<
1599    "\tunmarshTime: " << unmarshTotal << ". avg: " << unmarshTotal / (NUM_SEGMENTS *
         pInputC) <<
1600    endl <<
```

```
1601     "\tcopyHDTime: " <<  inputTotal - unmarshTotal << ". avg: " << (inputTotal -
             unmarshTotal) / (NUM_SEGMENTS * pInputC) <<
1602     endl <<
1603
1604     endl;
1605
1606 #endif
1607
1608 #if defined(TIME) || defined(TIME_FUN)
1609
1610 #undef DIFF_T0
1611
1612     delete counters;
1613     counters = NULL;
1614 #endif
1615
1616
1617     return ATOM(ok);
1618
1619 }
1620
1621 static ERL_NIF_TERM mapLD(ErlNifEnv * env, int argc, const ERL_NIF_TERM argv[]) {
1622
1623     //mapLD_S(Kernel::kernel(), InputList::[double()], OutputBuffer::deviceBuffer())
1624     OCL_INIT_CHECK()
1625     NIF_ARITY_CHECK(4)
1626
1627     /*get the parameters (Kernel::kernel(), InputList::[double()], OutputBuffer::
             deviceBuffer(), InputLength::non_neg_integer())****/
1628
1629     kernel_sync* pKernel_s = NULL;
1630     if (!enif_get_resource(env, argv[0], kernel_sync_rt, (void**) &pKernel_s)) {
1631         cerr << "ERROR :: mapLD_S: 1st parameter is not a kernel_sync" << endl ;
1632         return enif_make_badarg(env);
1633     }
1634
1635     ERL_NIF_TERM inputList = argv[1];
1636     if (!enif_is_list(env, argv[1])) {
1637         cerr << "ERROR :: mapLD_S: 2nd parameter is not a list" << endl ;
1638         return enif_make_badarg(env);
1639     }
1640
1641     cl_mem* pOutputBufferD = NULL;
1642     if (!enif_get_resource(env, argv[2], deviceBuffer_rt, (void**) &pOutputBufferD)
             || *pOutputBufferD == NULL) {
1643         cerr << "ERROR :: mapLD_S: 3nd parameter is not a device buffer" << endl ;
1644         return enif_make_badarg(env);
```

```
1645     }
1646
1647     uint inputListLength;
1648     if (!enif_get_uint(env, argv[3], &inputListLength)) {
1649
1650         cerr << "ERROR :: mapLL: 4th parameter is not a non_neg_integer()" << endl ;
1651         return enif_make_badarg(env);
1652     }
1653     /*****************************************************************************
         */
1654
1655
1656     uint pInputC = 1;
1657     ERL_NIF_TERM* pInputV[1] = { &inputList };
1658
1659 #ifdef DEBUG
1660     cerr << "DEBUG: mapLD" << endl;
1661 #endif
1662
1663     return mapLD_Impl_3Thread(env, pKernel_s,(void**) pInputV, pInputC,
         pOutputBufferD, inputListLength);
1664
1665 }
1666
1667
1668
1669 static ERL_NIF_TERM map2LD(ErlNifEnv * env, int argc, const ERL_NIF_TERM argv[]) {
1670
1671     //map2LD_S(Kernel::kernel(), InputList1::[double()], InputList2::[double()],
         OutputBuffer::deviceBuffer(), InputLength::non_neg_integer())
1672     OCL_INIT_CHECK()
1673     NIF_ARITY_CHECK(5)
1674
1675     /*get the parameters (Kernel::kernel(), InputList1::[double()], InputList2::[
         double()], OutputBuffer::deviceBuffer())***********/
1676
1677     kernel_sync* pKernel_s = NULL;
1678     if (!enif_get_resource(env, argv[0], kernel_sync_rt, (void**) &pKernel_s)) {
1679         cerr << "ERROR :: map2LD: 1st parameter is not a kernel_sync" << endl ;
1680         return enif_make_badarg(env);
1681     }
1682
1683     ERL_NIF_TERM inputLists[2];
1684
1685     inputLists[0] = argv[1];
1686     if (!enif_is_list(env, argv[1])) {
1687         cerr << "ERROR :: map2LD: 2nd parameter is not a list" << endl ;
```

```
1688            return enif_make_badarg(env);
1689        }
1690
1691        inputLists[1] = argv[2];
1692        if (!enif_is_list(env, argv[2])) {
1693            cerr << "ERROR :: map2LD: 2nd parameter is not a list" << endl ;
1694            return enif_make_badarg(env);
1695        }
1696
1697        cl_mem* pOutputBuffer = NULL;
1698        if (!enif_get_resource(env, argv[3], deviceBuffer_rt, (void**) &pOutputBuffer)
                || *pOutputBuffer == NULL) {
1699            cerr << "ERROR :: map2LD: 4th parameter is not a device buffer" << endl ;
1700            return enif_make_badarg(env);
1701        }
1702
1703        uint inputListLength;
1704        if (!enif_get_uint(env, argv[4], &inputListLength)) {
1705
1706            cerr << "ERROR :: mapLL: 5th parameter is not a non_neg_integer()" << endl ;
1707            return enif_make_badarg(env);
1708        }
1709        /*****************************************************************************
                */
1710
1711        uint pInputC = 2; //Map2
1712        ERL_NIF_TERM* pInputV[2] = { &(inputLists[0]), &(inputLists[1]) };
1713
1714 #ifdef DEBUG
1715        cerr << "DEBUG: map2LD"<< endl;
1716 #endif
1717
1718
1719        return mapLD_Impl_3Thread(env, pKernel_s,(void**) pInputV, pInputC,
                pOutputBuffer, inputListLength); //ok  | {error, Why}
1720 }
1721
1722
1723 static void* mapLL_unloadStage_Thread(void* obj) {
1724
1725 #ifdef DEBUG
1726        char msg[256];
1727 #endif
1728        MapLL_thread_params* params = (MapLL_thread_params *) obj;
1729
1730        ErlNifEnv* env = params->env; //needed by CLEANUP macros
1731        ErlNifMutex*& env_mtx = params->env_mtx;
```

```
1732
1733     Map_fun_counters*& counters = params->counters;
1734
1735     //locals
1736     size_t szInput_local = (params->szSegment / params->uiNumElemSegment) * params->
             uiNumElem;
1737
1738     cl_int ciErrNum;
1739
1740
1741
1742     uint zeroIdx;
1743     uint segmentOffset;
1744
1745     for (uint i_round = 0 ; i_round < params->numRounds; ++i_round) {
1746
1747         zeroIdx = (params->numQueues * i_round);
1748
1749 #ifdef DEBUG
1750         sprintf(msg, "DEBUG: Unload - Starting round[%d]\n", i_round);
1751         cerr << msg;
1752 #endif
1753
1754
1755         uint idx = 0;
1756         for (uint i_queue = 0 ; i_queue < params->numQueues; ++i_queue) {
1757
1758
1759 #ifdef DEBUG
1760             sprintf(msg, "DEBUG: Unload - %d - waiting to start round[%d]\n",
                     i_queue, i_round);
1761             cerr << msg;
1762 #endif
1763
1764             idx = zeroIdx + i_queue;
1765
1766             segmentOffset = idx * params->uiNumElemSegment;
1767
1768             cl_command_queue& curr_cmdQ = params->cmdQs[i_queue];
1769             cl_event& curr_run_done_evt = params->run_done_evt[i_queue][i_round];
1770
1771
1772
1773             //WAIT for RUN stage
1774             clWaitForEvents(1, &(curr_run_done_evt));
1775
1776             if(params->errorSignal) { //Someone signaled an error, exit.
```

```
1777 #ifdef DEBUG
1778                 sprintf(msg, "DEBUG: Unload - %d - Someone signaled an error, exit.\
                         n", i_queue);
1779                 cerr << msg;
1780 #endif
1781                 goto cleanup;
1782             }
1783
1784 #ifdef TIME
1785             GET_TIME(counters->round[i_round]->unload_copy_start[i_queue]);
1786 #endif
1787
1788             //********** D -> H ************
1789             double* pMappedOutputBufferSegmentH = params->pMappedOutputBufferH +
                     segmentOffset;
1790
1791             //blocking read because i need the data immediately for the marshalling
1792             THREAD_CHK_SUCCESS_CLEANUP_GOTO(
1793                 clEnqueueReadBuffer(curr_cmdQ, params->outputBufferD, CL_TRUE,
                         segmentOffset, params->szSegment, (void*)
                         pMappedOutputBufferSegmentH, 0, NULL, NULL); ,
1794
1795                 params->errorTerm ,
1796                 {
1797                   sprintf(msg, "DEBUG: Unload - %d - Error D -> H\n", i_queue); cerr
                           << msg;
1798                   params->signalError();
1799                 }
1800             )
1801             clFlush(curr_cmdQ);
1802
1803 #ifdef DEBUG
1804             sprintf(msg, "DEBUG: Unload - %d - D -> H\n", i_queue);
1805             cerr << msg;
1806 #endif
1807
1808
1809 #ifdef TIME
1810             GET_TIME(counters->round[i_round]->unload_copy_end[i_queue]);
1811 #endif
1812
1813             //overlap last marshalling with input buffer unmapping
1814             if(i_round == params->numRounds - 1 && i_queue == NUM_QUEUES-1)
1815                 clSetUserEventStatus(params->unmapInput_evt, CL_SUCCESS);
1816
1817
1818             //********** H -> L ************
```

```
1819
1820            ERL_NIF_TERM* outputTermsSegment = params->outputTerms + segmentOffset;
1821
1822            enif_mutex_lock(env_mtx);
1823
1824            for(uint i = 0; i < params->uiNumElemSegment; i++)
1825                outputTermsSegment[i] = enif_make_double(env,
                        pMappedOutputBufferSegmentH[i]);
1826
1827            enif_mutex_unlock(env_mtx);
1828
1829 #ifdef TIME
1830            GET_TIME((counters->round[i_round]->unload_marsh_end[i_queue]));
1831 #endif
1832
1833
1834
1835 #ifdef TIME
1836            GET_TIME(counters->round[i_round]->end[i_queue]);
1837 #endif
1838
1839 #ifdef DEBUG
1840            sprintf(msg, "DEBUG: Unload - %d - H -> L\nDEBUG: %d - Round END\n",
                    i_queue, i_queue);
1841            cerr << msg;
1842 #endif
1843
1844        }//end queue
1845
1846    }//end round
1847
1848    //signal master to start list creation, overlaps with the following unmap
1849    clSetUserEventStatus(params->lastMarshalling_evt, CL_SUCCESS);
1850
1851 cleanup:
1852
1853    if(params->errorSignal) {//release every lock before exiting to avoid deadlocks
1854
1855 #ifdef DEBUG
1856        sprintf(msg, "DEBUG: Unload - Error detected, set all events\n");
1857        cerr << msg;
1858 #endif
1859
1860        clSetUserEventStatus(params->unmapInput_evt, CL_SUCCESS);
1861        clSetUserEventStatus(params->lastMarshalling_evt, CL_SUCCESS);
1862    }
1863
```

```
1864     //UNMAP using last queue
1865     clEnqueueUnmapMemObject(params->cmdQs[NUM_QUEUES-1], params->outputBufferH,
             params->pMappedOutputBufferH, 0, NULL, NULL);
1866
1867 #ifdef TIME
1868     GET_TIME(counters->unmapOutput_end);
1869 #endif
1870
1871     //**********************RELEASE BUFFERS*******************
1872     // outputBufferH szInput
1873     clReleaseMemObject(params->outputBufferH);
1874
1875 #ifdef TIME
1876     GET_TIME(counters->releaseOutBufferH_T);
1877 #endif
1878
1879     //outputBufferD szInput
1880     clReleaseMemObject(params->outputBufferD);
1881 #ifdef TIME
1882     GET_TIME(counters->releaseOutBufferD_T);
1883 #endif
1884
1885
1886 #ifdef TIME
1887     GET_TIME((counters->releaseInBuffersD_T_start));
1888 #endif
1889     //inputBuffersD[pInputC][NUM_SEGMENTS] * szSegment = szInput
1890     for (uint i_input = 0; i_input <  params->pInputC; ++i_input) {
1891
1892         for (int i_segm = 0; i_segm <  params->numSegments; ++i_segm) {
1893             clReleaseMemObject(params->inputBuffersD[i_input][i_segm]);
1894         }
1895
1896         delete [] params->inputBuffersD[i_input];
1897     }
1898 #ifdef TIME
1899     GET_TIME((counters->releaseInBuffersD_T_end));
1900 #endif
1901
1902
1903 #ifdef DEBUG
1904     sprintf(msg, "DEBUG: Unload - Done\n");
1905     cerr << msg;
1906 #endif
1907
1908     return NULL;
1909 }
```

```
1910
1911
1912 //Number of segments of list splitting in mapLL
1913 #define NUM_SEGM 8
1914
1915 static ERL_NIF_TERM mapLL_Impl_3Thread(ErlNifEnv * env, kernel_sync* pKernel_s, void
         ** _pInputV, uint pInputC, void* _pOutputBuffer, uint listLength) {
1916
1917 #ifdef DEBUG
1918     cerr << "DEBUG: M - mapLL_Impl_3Thread" << endl;
1919 #endif
1920
1921     ERL_NIF_TERM toReturn;
1922
1923     const uint NUM_SEGMENTS = NUM_SEGM;
1924
1925     const uint NUM_QUEUES_LOCAL = NUM_QUEUES;
1926
1927     const uint numRounds = NUM_SEGMENTS/NUM_QUEUES_LOCAL;
1928
1929
1930 #if defined(TIME) || defined(TIME_FUN)
1931
1932 #define DIFF_T0(COUNTER) nsec2usec(diff( &(counters->fun_start), &(COUNTER)))
1933
1934     Map_fun_counters* counters = new Map_fun_counters(numRounds, pInputC);
1935 #endif
1936
1937 #ifdef TIME
1938
1939     timespec createEventsT_start, createEventsT_end,
1940     createInputBuffersH[pInputC],
1941     createInputBuffersD[pInputC],
1942
1943     createOutputBufferD;
1944
1945     timespec joinT[3],
1946     list_createdT_start, list_createdT_end,
1947     releaseEventsT;
1948
1949
1950 #endif
1951
1952 #if defined(TIME) || defined(TIME_FUN)
1953     GET_TIME(counters->fun_start);
1954 #endif
1955
```

```
1956
1957     std::vector<cl_event> events;
1958
1959     //cast input/output to type relevant here
1960     //pInputV is an array of ERL_NIF_TERM* representing, each one, an erlang list
1961     ERL_NIF_TERM** inputLists = (ERL_NIF_TERM**) _pInputV;
1962
1963     //input lists must have at least minListLen elements,
1964     //otherwise they can't be split in NUM_SEGMENTS segments
1965     uint listLen = listLength;
1966
1967
1968     //check buffer allignement requirements (depends on the device)
1969     uint minListLen = device_minBaseAddrAlignByte / sizeof(double);
1970     if(listLen/NUM_SEGMENTS < minListLen)
1971         return make_error(env, ATOM(skel_ocl_input_list_too_short));
1972
1973
1974     uint uiNumElem = listLen;
1975
1976     uint uiNumElemSegment = uiNumElem / NUM_SEGMENTS;
1977
1978     size_t szInput = uiNumElem * sizeof(double);
1979
1980     size_t szSegment = szInput / NUM_SEGMENTS;
1981
1982
1983 #ifdef DEBUG
1984     cerr << "DEBUG: M - mapLL_impl:" << "#Elem: "<< uiNumElem << " #ElemSegment: "<<
          uiNumElemSegment <<" szInput: "<< szInput << " szSegment: "<< szSegment <<
         endl;
1985 #endif
1986
1987     cl_int ciErrNum = 0;
1988
1989     //working queues
1990     cl_command_queue cmdQs[2] = {getCommandQueue(0), getCommandQueue(1)};
1991
1992     cl_context context = getContext();
1993
1994     //mutex protecting env from concurrent modifications (enif_make_xxx)
1995     ErlNifMutex* env_mtx = enif_mutex_create((char*)"env_mtx");
1996
1997     //terms array from which to create the output list using
          enif_make_list_from_array, filled by unload stage
1998     ERL_NIF_TERM* outputTerms = NULL;
1999
```

```
2000
2001 #ifdef TIME
2002     GET_TIME(createEventsT_start);
2003 #endif
2004
2005     //********events setup [NUM_QUEUES][numRounds])
2006     cl_event* run_start_evt[NUM_QUEUES_LOCAL];
2007     cl_event* run_done_evt[NUM_QUEUES_LOCAL];
2008
2009     for (int i = 0; i < NUM_QUEUES_LOCAL; ++i) {
2010         run_start_evt[i] = new cl_event[numRounds];
2011         run_done_evt[i] = new cl_event[numRounds];
2012
2013         for (int i_round = 0; i_round < numRounds; ++i_round) {
2014             run_start_evt[i][i_round] = clCreateUserEvent(context, &ciErrNum);
2015             run_done_evt[i][i_round] = clCreateUserEvent(context, &ciErrNum);
2016
2017             events.push_back(run_start_evt[i][i_round]);
2018             events.push_back(run_done_evt[i][i_round]);
2019
2020         }
2021     }
2022
2023     //to allow overlapping between unmarshalling and mapping the outputBuffer (can't
          map more in parallel)
2024     // fired by load stage after finishing the mapping of inputBuffer, wait by
          unload
2025     cl_event mapOutput_evt = clCreateUserEvent(context, &ciErrNum);
2026     events.push_back(mapOutput_evt);
2027
2028     //to allow overlapping between input buffer unmapping and last marshalling (set
          by unload, wait by load).
2029     cl_event unmapInput_evt = clCreateUserEvent(context, &ciErrNum);
2030     events.push_back(unmapInput_evt);
2031
2032     //to overlap output list creation to unmapOutputBuffer (set by unload, wait by
          master)
2033     cl_event lastMarshalling_evt = clCreateUserEvent(context, &ciErrNum);
2034     events.push_back(lastMarshalling_evt);
2035
2036 #ifdef TIME
2037     GET_TIME(createEventsT_end);
2038 #endif
2039
2040     //*********threads setup: for now, one thread per stage.
2041     //Load, run, unload. Each stage works on 2 command queues
2042     const uint NUM_STAGES = 3;
```

```
2043     ErlNifTid tid[NUM_STAGES];
2044
2045     ErlNifTid& loadStage = tid[0];
2046     ErlNifTid& runStage = tid[1];
2047     ErlNifTid& unloadStage = tid[2];
2048
2049     //object holding everything needed by the threads
2050     MapLL_thread_params* conf = NULL;
2051
2052
2053     //***********for each input list, create:
2054     // 1 inputBufferH
2055     // NUM_SEGMENTS inputBufferD
2056
2057     cl_mem inputBuffersH[pInputC];
2058
2059     //[pInputC][NUM_SEGMENTS]
2060     cl_mem* inputBuffersD[pInputC];
2061     for (int i = 0; i < pInputC; ++i)
2062         inputBuffersD[i] = new cl_mem[NUM_SEGMENTS];
2063
2064
2065     //reuse first inputBuffersH as output buffer
2066
2067     cl_mem outputBufferH;
2068
2069     // device output buffer (same size as input)
2070     cl_mem outputBufferD;
2071
2072
2073     for(uint i_list = 0; i_list < pInputC; ++i_list) {
2074         // create one input host buffer of size szInput, released by Unload
2075         CHK_SUCCESS_CLEANUP_GOTO(
2076                                     createBuffer(szInput, CL_MEM_READ_WRITE |
                                            CL_MEM_ALLOC_HOST_PTR, &(inputBuffersH[i_list])
                                            ); ,
2077                                     ;
2078                                     )
2079
2080 #ifdef TIME
2081         GET_TIME(createInputBuffersH[i_list]);
2082 #endif
2083
2084
2085 #ifdef DEBUG
2086         cerr << "DEBUG: M - " << "inputBuffersH[" << i_list << "] CREATED: size: "
                << szInput << endl;
```

```
2087 #endif
2088
2089
2090         //*********create NUM_SEGMENTS input device buffers having size szSegment
                 *********
2091         //released by Unload
2092         for (int i_segm = 0; i_segm < NUM_SEGMENTS; ++i_segm) {
2093
2094             CHK_SUCCESS_CLEANUP_GOTO(
2095                                         createBuffer(szSegment, CL_MEM_READ_ONLY, &(
                                                inputBuffersD[i_list][i_segm]) ); ,
2096                                         ;
2097                                         )
2098
2099
2100 #ifdef TIME
2101             GET_TIME(createInputBuffersD[i_list]);
2102 #endif
2103
2104 #ifdef DEBUG
2105             cerr << "DEBUG: M - " << "inputBuffersD["<<i_list <<"]["  <<i_segm<<"]
                    CREATED: size: " << szSegment<< endl;
2106 #endif
2107         }
2108 #ifdef DEBUG
2109         cerr << "DEBUG: M - " << "inputBuffersD CREATED"<< endl;
2110 #endif
2111
2112     }
2113
2114     outputBufferH = inputBuffersH[0];//reuse first inputBufferH as outputBufferH
2115     clRetainMemObject(outputBufferH);
2116
2117     //*********Create one OUTPUT buffer on DEVICE*********
2118     //released by Unload
2119     CHK_SUCCESS_CLEANUP_GOTO(
2120                                 createBuffer(szInput, CL_MEM_WRITE_ONLY, &(
                                        outputBufferD) ); ,
2121                                 ;
2122                                 )
2123
2124 #ifdef TIME
2125     GET_TIME(createOutputBufferD);
2126 #endif
2127
2128
```

```
2129    //****Create the array that will store the ERL_NIF_TERM representing the output
            elements*********
2130
2131    outputTerms =
2132    (ERL_NIF_TERM*) enif_alloc(sizeof(ERL_NIF_TERM) * uiNumElem);
2133
2134    if(outputTerms == NULL) {
2135        cerr << "DEBUG: M - " << "enif_alloc returned NULL."<< ciErrNum << endl;
2136
2137        toReturn = make_error(env, ATOM(erl_enomem));
2138        goto cleanup;
2139    }
2140
2141
2142
2143    //Clone kernel, 'cause i need two of them, one per queue
2144    cl_kernel kernels[NUM_QUEUES_LOCAL];
2145
2146    kernels[0] = pKernel_s->kernel;
2147
2148    if(NUM_QUEUES_LOCAL == TWO)
2149        cloneKernel(pKernel_s->kernel, &kernels[1]);
2150
2151    if(NUM_QUEUES_LOCAL > MAX_QUEUES) {
2152        toReturn = make_error(env, ATOM(ocl_num_queue_not_supported));
2153        goto cleanup;
2154    }
2155
2156
2157
2158    //**************************Main LOOP************************
2159
2160    //iterator on the list, points to the head of the current segment of the list
2161    ERL_NIF_TERM currList[pInputC];
2162    for (uint i = 0; i < pInputC; ++i)
2163        currList[i] = *(inputLists[i]);
2164
2165
2166    conf = new MapLL_thread_params (
2167                                    mapOutput_evt,
2168                                    unmapInput_evt,
2169                                    lastMarshalling_evt,
2170                                    run_start_evt, run_done_evt,
2171                                    env, env_mtx,
2172                                    pInputC, cmdQs, kernels,
2173                                    inputBuffersD, outputBufferD,
2174                                    inputBuffersH,
```

```
2175                                        outputBufferH,
2176                                        currList,
2177 #ifdef TIME
2178                                        counters,
2179 #else
2180                                        NULL,
2181 #endif
2182                                        numRounds,
2183                                        NUM_QUEUES_LOCAL,
2184                                        0,
2185                                        uiNumElem, uiNumElemSegment, szSegment,
2186                                        outputTerms
2187                                        );
2188
2189
2190
2191 #ifdef DEBUG
2192     cerr << "DEBUG: M - Starting threads." << endl;
2193 #endif
2194
2195     enif_thread_create((char*) "mapLL loadStage", &loadStage, mapLL_loadStage_Thread
          , conf, NULL);
2196
2197     enif_thread_create((char*) "mapLL runStage", &runStage, mapLL_runStage_Thread,
          conf, NULL);
2198
2199     enif_thread_create((char*) "mapLL unloadStage", &unloadStage,
          mapLL_unloadStage_Thread, conf, NULL);
2200
2201
2202 #ifdef TIME
2203     GET_TIME((counters->fun_prologue));
2204 #endif
2205
2206
2207     //wait last marshalling then start creating the list
2208     clWaitForEvents(1, &conf->lastMarshalling_evt);
2209
2210
2211     if(conf->errorSignal) {
2212         //something's gone wrong, cleanup and return the error term (the last one
              generated)
2213 #ifdef DEBUG
2214         cerr << "DEBUG: M - Got an error: cleanup and exit"<< endl;
2215 #endif
2216
2217         toReturn = conf->errorTerm;
```

```
2218     }
2219     else {
2220         //everything went fine
2221 #ifdef DEBUG
2222         cerr << "DEBUG: M - Creating output list\n";
2223 #endif
2224 
2225 
2226 #ifdef TIME
2227         GET_TIME(list_createdT_start);
2228 #endif
2229 
2230         toReturn = enif_make_tuple2(env,ATOM(ok),enif_make_list_from_array(env,
                outputTerms, uiNumElem));
2231 
2232 #ifdef TIME
2233         GET_TIME(list_createdT_end);
2234 #endif
2235     }
2236 
2237 
2238     //wait for termination
2239     for(uint i = 0; i < NUM_STAGES; ++i) {
2240 
2241         enif_thread_join(tid[i], NULL);
2242 
2243 #ifdef TIME
2244         GET_TIME((joinT[i]));
2245 #endif
2246 
2247 #ifdef DEBUG
2248         char msg[64];
2249         sprintf(msg, "DEBUG: M - Thread %d joined\n",i);
2250         cerr << msg;
2251 #endif
2252     }
2253 
2254     //check again after joining, shouldn't be useful since list creation starts
            after the last marshalling, just before outputBuffer's unmapping,
2255     //at the very end of the computation when nobody can signal an error anymore.
2256     if(conf->errorSignal) {
2257         //something's gone wrong, cleanup and return the error term (the last one
                generated)
2258 #ifdef DEBUG
2259         cerr << "DEBUG: M - Got an error: cleanup and exit"<< endl;
2260 #endif
2261         toReturn = conf->errorTerm;
```

```
2262     }
2263
2264     //*****************CLEANUP label*****************
2265 cleanup:
2266
2267     if(outputTerms) {
2268         enif_free(outputTerms);//2-3 usecs
2269         outputTerms = NULL;
2270     }
2271
2272
2273     //*****************EVENTS*****************
2274     if(!events.empty())
2275         releaseEvents(events);
2276
2277     for(uint i = 0; i < NUM_QUEUES_LOCAL; i++){
2278         delete [] run_start_evt[i];
2279         delete [] run_done_evt[i];
2280     }
2281 #ifdef TIME
2282     GET_TIME((releaseEventsT));
2283 #endif
2284
2285     //*****************KERNELS*****************
2286     if(NUM_QUEUES_LOCAL == 2){
2287         if(kernels[1])
2288             clReleaseKernel(kernels[1]); //2 usecs
2289         kernels[1] = NULL;
2290     }
2291
2292     //*****************ENV MUTEX*****************
2293     if(env_mtx)
2294         destroyMutexes(&env_mtx, 1);//1 usec
2295
2296     //*****************CONF*****************
2297     if(conf)
2298         delete conf;
2299
2300
2301 #ifdef DEBUG
2302     cerr << "DEBUG: M - Cleanup done. Return"<< endl;
2303 #endif
2304
2305
2306 #if defined(TIME) || defined(TIME_FUN)
2307     GET_TIME((counters->fun_end));
2308 #endif
```

```
2309
2310
2311
2312     //**********************RUN STATISTITCS*******************
2313
2314
2315 #if defined(TIME) || defined(TIME_FUN)
2316
2317     cerr << endl <<
2318     "------------------------------------ map"<< pInputC <<"LL: (usec)
            ------------------------------------" <<
2319     endl <<
2320     "Processing " << pInputC << " X " << uiNumElem <<" elements in "<< NUM_SEGMENTS
            <<" segments (" << uiNumElemSegment <<" per segment)"<<
2321     endl <<
2322     "Total run time: " << DIFF_T0(counters->fun_end) <<
2323     endl;
2324
2325 #endif
2326
2327
2328 #ifdef TIME
2329
2330     long prologueTime = DIFF_T0(counters-> fun_prologue);
2331
2332
2333     cerr << endl <<
2334     "Function prologue time: " <<  prologueTime << "\t\t\t\t\tT: "<< prologueTime <<
2335     endl;
2336
2337     char str[128];
2338     sprintf(str, "\tcreateEvents: %lu \t\t\t\t\tT: %lu\n",  DIFF(createEventsT_start
            , createEventsT_end), DIFF_T0( createEventsT_end)); cerr << str;
2339
2340     for (uint i_input = 0; i_input < pInputC; i_input++) {
2341         sprintf(str, "\tcreateInputBuffersH[%d]: \t\t\t\tT: %lu\n", i_input, DIFF_T0
                ( createInputBuffersH[i_input])); cerr << str;
2342
2343         sprintf(str,
2344                 "Load - mapInputBuffers[%d]: %lu \t\t\t\tT: %lu\n", i_input,
2345                 i_input == 0 ?
2346                 DIFF( counters->mapInput_start, counters->mapInput_end[i_input]) :
                        DIFF( counters->mapInput_end[i_input-1], counters->mapInput_end[
                        i_input]),
2347                 DIFF_T0( counters->mapInput_end[i_input])
2348                 );
2349         cerr << str;
```

```
2350
2351        sprintf(str, "\tcreateInputBuffersD[%d]: \t\t\t\tT: %lu\n", i_input, DIFF_T0
                ( createInputBuffersD[i_input])); cerr << str;
2352    }
2353
2354
2355    sprintf(str, "\tcreateOutputBufferD: \t\t\t\tT: %lu\n",  DIFF_T0(
            createOutputBufferD)); cerr << str;
2356
2357
2358    uint kernelTotal = 0, inputTotal = 0, outputTotal = 0, marshTotal = 0,
            unmarshTotal = 0;
2359    uint i_round;
2360    //round statistics
2361    for (i_round = 0; i_round < numRounds; i_round++) {
2362        uint tKernel[2] = {0,0};
2363        uint tInput = 0, tOutput = 0;
2364        uint tStartRound[2];
2365
2366        cerr <<
2367        "\nRound["<< i_round << "] total time: " << DIFF(( counters->round[i_round
                ]->start[0]), ( counters->round[i_round]->end[NUM_QUEUES_LOCAL-1]))<<
2368        endl;
2369
2370
2371        for (int i_queue = 0; i_queue < NUM_QUEUES_LOCAL; i_queue++) {
2372
2373            inputTotal +=
2374            tInput = DIFF(( counters->round[i_round]->load_start[i_queue]), (
                    counters->round[i_round]->load_end[i_queue]));
2375
2376
2377            tStartRound[i_queue] = DIFF_T0(counters->round[i_round]-> start[i_queue
                    ]);
2378            cerr <<
2379            "\n\t"<< i_queue << " - R["<< i_round << "] Start round\t\t\t\tT: " <<
                     tStartRound[i_queue]<<
2380            endl <<
2381            "\t"<< i_queue<< " - R["<< i_round << "] Load total time: " << tInput <<
2382            endl;
2383
2384            for (uint i_input = 0; i_input < pInputC; i_input++) {
2385                long unmarsh = 0, t_Unmarsh;
2386                long cpDH_0 = 0, t_cpDH_0;
2387
2388                unmarshTotal +=
```

```
2389                unmarsh = DIFF((counters->round[i_round]->load_input_start[i_queue][
                        i_input]), (counters->round[i_round]->load_unmarsh_end[i_queue][
                        i_input]));
2390
2391                t_Unmarsh = DIFF_T0(counters->round[i_round]-> load_unmarsh_end[
                        i_queue][i_input]);
2392
2393
2394                cpDH_0 = DIFF((counters->round[i_round]->load_unmarsh_end[i_queue][
                        i_input]), (counters->round[i_round]->load_copyHD_end[i_queue][
                        i_input]));
2395                t_cpDH_0 = DIFF_T0(counters->round[i_round]-> load_copyHD_end[
                        i_queue][i_input]);
2396
2397                cerr <<
2398                "\t\t"<< i_queue << " - R["<< i_round << "] Input["<< i_input<<"]
                        unmarshall: " << unmarsh << "\t\tT: "<<  t_Unmarsh <<
2399                endl <<
2400                "\t\t"<< i_queue << " - R["<< i_round << "] Input["<< i_input<<"] H
                        -> D: " << cpDH_0 << "\t\t\tT: "<<  t_cpDH_0 <<
2401                endl;
2402            }
2403
2404
2405
2406            long t_Kernel = DIFF_T0(counters->round[i_round]-> run_end[i_queue]);
2407            kernelTotal +=
2408            tKernel[i_queue] = DIFF((counters->round[i_round]->run_start[i_queue]),
                    (counters->round[i_round]->run_end[i_queue]));
2409
2410            cerr <<
2411            "\t" << i_queue<< " - R["<< i_round << "] KERNEL computation time: " <<
                    tKernel[i_queue] << "\t\t\tT: "<<  t_Kernel <<
2412            endl;
2413
2414
2415
2416            long marsh, cpDH;
2417            long t_marsh, t_cpDH;
2418
2419            cpDH = DIFF((counters->round[i_round]->unload_copy_start[i_queue]), (
                    counters->round[i_round]->unload_copy_end[i_queue]));
2420            t_cpDH = DIFF_T0(counters->round[i_round]->unload_copy_end[i_queue]);
2421
2422            marshTotal +=
2423            marsh = DIFF((counters->round[i_round]->unload_copy_end[i_queue]), (
                    counters->round[i_round]->unload_marsh_end[i_queue]));
```

```
2424            t_marsh = DIFF_T0(counters->round[i_round]->unload_marsh_end[i_queue]);
2425
2426
2427            //sprintf(msg, "\t%d - R[%d] Unload time: %li", i_queue, i_round, cpDH +
                    marsh);
2428            cerr <<
2429            "\t"<< i_queue << " - R["<< i_round << "] Unload time: " << cpDH + marsh
                    <<
2430            endl <<
2431            "\t\t"<< i_queue << " - R[" << i_round << "] D -> H: " << cpDH << "\t\t\
                    t\tT: "<<  t_cpDH <<
2432            endl <<
2433            "\t\t"<< i_queue << " - R["<< i_round << "] marshall: " << marsh << "\t\
                    t\tT: "<<  t_marsh <<
2434            endl;
2435
2436        }// end queues
2437    }//end rounds
2438
2439    //FROM last load (lastRound, last_queue) TO unmapInput_end
2440    cerr << "Load - unmapInputBuffers: "<< DIFF(counters->unmapInput_start, counters
            ->unmapInput_end) <<
2441    "\t\t\t\t\tT: " << DIFF_T0( counters->unmapInput_end)
2442    << endl;
2443
2444    cerr << "Load - releaseInBuffersH: " << DIFF(counters->unmapInput_end, counters
            ->releaseInBuffersH_T) <<"\t\t\t\t\tT: " << DIFF_T0( counters->
            releaseInBuffersH_T) << "\n";
2445
2446
2447    cerr << "Unload - unmapOutputBuffers: "<< DIFF(counters->round[numRounds-1]->end
            [NUM_QUEUES_LOCAL-1], counters->unmapOutput_end) <<
2448    "\t\t\t\tT: " << DIFF_T0( counters->unmapOutput_end)
2449    << endl;
2450
2451    cerr << "Unload - releaseOutBufferH: " << DIFF(counters->unmapOutput_end,
            counters->releaseOutBufferH_T) <<"\t\t\t\tT: " << DIFF_T0(counters->
            releaseOutBufferH_T) << "\n";
2452
2453    cerr << "Unload - releaseOutBufferD: " << DIFF( counters->releaseOutBufferH_T,
            counters->releaseOutBufferD_T) <<"\t\t\t\tT: " << DIFF_T0(counters->
            releaseOutBufferD_T) << "\n";
2454
2455    cerr << "Unload - releaseInBuffersD: " << DIFF(counters->
            releaseInBuffersD_T_start, counters->releaseInBuffersD_T_end) <<"\t\t\t\tT:
            " << DIFF_T0(counters->releaseInBuffersD_T_end) << "\n";
2456
```

```
2457
2458     cerr << "M - make_list: "<<  DIFF(list_createdT_start, list_createdT_end)<< "\t\
             t\t\t\t\tT: " << DIFF_T0( list_createdT_end) << "\n";
2459
2460
2461     for(int i = 0; i < NUM_STAGES; i++)
2462         cerr << "M - join "<< i << "\t\t\t\t\t\t\tT: " << DIFF_T0( joinT[i]) << "\n"
                 ;
2463
2464
2465     cerr << "M - releaseEvents: "<< DIFF(joinT[NUM_STAGES-1], releaseEventsT ) << "\
             t\t\t\t\t\tT: " << DIFF_T0( releaseEventsT) << "\n";
2466
2467     cerr << endl <<
2468     "KERNEL total time: " << kernelTotal << ". avg on "<< numRounds <<" rounds: " <<
          kernelTotal/ NUM_SEGMENTS <<
2469     endl <<
2470     "Load time: " << inputTotal << ". avg: " << inputTotal / NUM_SEGMENTS <<
2471     endl <<
2472     "\tunmarshTime: " << unmarshTotal << ". avg: " << unmarshTotal / (NUM_SEGMENTS *
          pInputC) <<
2473     endl <<
2474     "\tcopyHDTime: " <<  inputTotal - unmarshTotal << ". avg: " << (inputTotal -
          unmarshTotal) / (NUM_SEGMENTS * pInputC) <<
2475     endl <<
2476     "\tmarshTime: " << marshTotal << ". avg: " << (marshTotal) / NUM_SEGMENTS <<
2477     endl;
2478
2479 #endif
2480
2481 #if defined(TIME) || defined(TIME_FUN)
2482
2483 #undef DIFF_T0
2484     delete counters;
2485     counters = NULL;
2486 #endif
2487
2488
2489     return toReturn; //return list
2490 }
2491
2492
2493 static ERL_NIF_TERM mapLL(ErlNifEnv * env, int argc, const ERL_NIF_TERM argv[]) {
2494
2495     OCL_INIT_CHECK()
2496     const uint NUM_ARGS = 3;
2497     //mapLL(Kernel::kernel(), InputList::[double()], InputLength::non_neg_integer())
```

```
2498
2499     NIF_ARITY_CHECK(NUM_ARGS)
2500
2501     /*get the parameters (Kernel::kernel(), InputList::[double()], InputLength::
             non_neg_integer()****/
2502
2503     kernel_sync* pKernel_s = NULL;
2504     if (!enif_get_resource(env, argv[0], kernel_sync_rt, (void**) &pKernel_s)) {
2505         cerr << "ERROR :: mapLL: 1st parameter is not a kernel_sync" << endl ;
2506         return enif_make_badarg(env);
2507     }
2508
2509     ERL_NIF_TERM inputList = argv[1];
2510     if (!enif_is_list(env, argv[1])) {
2511         cerr << "ERROR :: mapLL: 2nd parameter is not a list" << endl ;
2512         return enif_make_badarg(env);
2513     }
2514
2515
2516     uint inputListLength;
2517     if (!enif_get_uint(env, argv[2], &inputListLength)) {
2518
2519         cerr << "ERROR :: mapLL: 3nd parameter is not a non_neg_integer()" << endl ;
2520         return enif_make_badarg(env);
2521     }
2522
2523     /*****************************************************************************
             */
2524
2525
2526     uint pInputC = 1;
2527     ERL_NIF_TERM* pInputV[1] = { &inputList };
2528
2529 #ifdef DEBUG
2530     cerr << "DEBUG: mapLL" << endl;
2531 #endif
2532
2533     return mapLL_Impl_3Thread(env,pKernel_s, (void**) pInputV, pInputC, NULL,
             inputListLength);
2534
2535 }
2536
2537
2538 static ERL_NIF_TERM map2LL(ErlNifEnv * env, int argc, const ERL_NIF_TERM argv[]) {
2539
2540     //map2LL_S(Kernel::kernel(), InputList1::[double()], InputList2::[double()])
2541     OCL_INIT_CHECK()
```

```
2542     const int ARGS_NUM = 4;
2543 
2544     NIF_ARITY_CHECK(ARGS_NUM)
2545 
2546     /*get the parameters (Kernel::kernel(), InputList::[double()]****/
2547 
2548     kernel_sync* pKernel_s = NULL;
2549     if (!enif_get_resource(env, argv[0], kernel_sync_rt, (void**) &pKernel_s)) {
2550         cerr << "ERROR :: map2LL: 1st parameter is not a kernel_sync" << endl ;
2551         return enif_make_badarg(env);
2552     }
2553 
2554     ERL_NIF_TERM inputList1 = argv[1];
2555     if (!enif_is_list(env, argv[1])) {
2556         cerr << "ERROR :: map2LL: 2nd parameter is not a list" << endl ;
2557         return enif_make_badarg(env);
2558     }
2559 
2560     ERL_NIF_TERM inputList2 = argv[2];
2561     if (!enif_is_list(env, argv[2])) {
2562         cerr << "ERROR :: map2LL: 3rd parameter is not a list" << endl ;
2563         return enif_make_badarg(env);
2564     }
2565 
2566     uint inputListLength;
2567     if (!enif_get_uint(env, argv[3], &inputListLength)) {
2568 
2569         cerr << "ERROR :: map2LL: 4th parameter is not a non_neg_integer()" << endl
                 ;
2570         return enif_make_badarg(env);
2571     }
2572     /*************************************************************************
         */
2573 
2574 
2575     uint pInputC = 2;
2576     ERL_NIF_TERM* pInputV[2] = { &inputList1, &inputList2 };
2577 
2578 #ifdef DEBUG
2579     cerr << "DEBUG: map2LL" << endl;
2580 #endif
2581 
2582     return mapLL_Impl_3Thread(env, pKernel_s, (void**) pInputV, pInputC, NULL,
         inputListLength);
2583 
2584 
2585 }
```

B.2.3 Reduce Skeleton NIFs: `reduce_nifs.cpp`

```
1  /**********************   REDUCE   ****************************************/
2
3
4  static ERL_NIF_TERM reduceDD_Impl(ErlNifEnv * env, kernel_sync* pKernel_s, cl_mem*
       pInputBuffer, cl_mem* pOutputBuffer) {
5
6      // %%reduceDD(Kernel::kernel(), InputBuffer::deviceBuffer(), OutputBuffer::
           deviceBuffer())
7
8      //Works with modified reduce6 kernel
9
10     //Max values from NVIDIA reduce example
11     const size_t maxThreads = 256;
12     const size_t maxBlocks = 64;
13
14     size_t uiNumBlocks = 0;
15     size_t uiNumThreads = 0;
16
17     size_t globalWorkSize[1];
18     size_t localWorkSize[1];
19
20     //********Check buffers dimensions: Input must be power of 2
21     size_t szInputBufferByte = 0;
22
23     CHK_SUCCESS(getBufferSizeByte(*pInputBuffer, &szInputBufferByte);)
24
25     uint uiN = szInputBufferByte / sizeof(double); //problem size is a power of 2
26
27     getNumBlocksAndThreads(uiN, maxBlocks,maxThreads, uiNumBlocks, uiNumThreads);
28
29     //allocate device output buffer having szOutputBufferByte size
30     size_t szOutputBufferByte = sizeof(double) * uiNumBlocks ; //one value per block
31
32     cl_mem outputBufferD = NULL;
33
34     CHK_SUCCESS(createBuffer(szOutputBufferByte, CL_MEM_READ_WRITE, &outputBufferD)
           ;)
35
36     // Decide size of shared memory (one value per thread but kernel needs at least
           64*sizeof(double) bytes)
37     size_t szLocalMemByte = (uiNumThreads <= 32) ? 2 * uiNumThreads * sizeof(double)
            : uiNumThreads * sizeof(double);
38
39
40     //*********set reduce kernel parameters (first reduction)**********************
```

```
41      cl_int ciErrNum = 0;
42
43      // clSetKernelArg is not thread-safe
44      enif_mutex_lock(pKernel_s->mtx);
45
46      ciErrNum |= clSetKernelArg(pKernel_s->kernel, 0, sizeof(cl_mem), (void*)
            pInputBuffer);
47      ciErrNum |= clSetKernelArg(pKernel_s->kernel, 1, sizeof(cl_mem), (void*) &
            outputBufferD);
48      ciErrNum |= clSetKernelArg(pKernel_s->kernel, 2, sizeof(unsigned int), (void*) &
            uiN);
49      ciErrNum |= clSetKernelArg(pKernel_s->kernel, 3, szLocalMemByte, NULL);
50
51      enif_mutex_unlock(pKernel_s->mtx);
52
53      CHK_SUCCESS_CLEANUP(ciErrNum; , clReleaseMemObject(outputBufferD);)
54
55
56      //****First reduce all elements so that each block produces one element
            **********
57      globalWorkSize[0] = uiNumThreads * uiNumBlocks;
58      localWorkSize[0]  = uiNumThreads;
59
60
61      cl_command_queue cmdQ_0 = getCommandQueue(0);
62
63      //event which the second round awaits, needed to express the dependency that
            exists between the two rounds.
64      //It's mandatory only when using an out-of-order queue.
65      cl_event firstRoundEvt = NULL;
66
67      CHK_SUCCESS_CLEANUP(
68                          clEnqueueNDRangeKernel(cmdQ_0, pKernel_s->kernel, 1, NULL,
                                &(globalWorkSize[0]), &(localWorkSize[0]), 0, NULL, &
                                firstRoundEvt); ,
69                          clReleaseMemObject(outputBufferD);
70
71                          )
72
73      //*********Second reduction**********************
74      uiN = uiNumBlocks;
75
76      //since problem size (uiN) has changed, recompute uiNumThreads and
            szLocalMemByte
77      getNumBlocksAndThreads(uiN, maxBlocks, maxThreads, uiNumBlocks, uiNumThreads);
78      szLocalMemByte = (uiNumThreads <= 32) ? 2 * uiNumThreads * sizeof(double) :
            uiNumThreads * sizeof(double);
```

```
79
80      enif_mutex_lock(pKernel_s->mtx);
81
82      //use as input previous reduction's output buffer
83      ciErrNum |= clSetKernelArg(pKernel_s->kernel, 0, sizeof(cl_mem), (void*) &
            outputBufferD);
84      ciErrNum |= clSetKernelArg(pKernel_s->kernel, 1, sizeof(cl_mem), (void*) &
            outputBufferD);
85      ciErrNum |= clSetKernelArg(pKernel_s->kernel, 2, sizeof(unsigned int), (void*) &
            uiN);
86      ciErrNum |= clSetKernelArg(pKernel_s->kernel, 3, szLocalMemByte, NULL);
87
88      enif_mutex_unlock(pKernel_s->mtx);
89
90      CHK_SUCCESS_CLEANUP(ciErrNum; , clReleaseMemObject(outputBufferD);)
91
92
93      //**********compute the result scalar from numBlock vector*********************
94      globalWorkSize[0] = 1 * uiNumThreads; //only one block left
95      localWorkSize[0]  = uiNumThreads;
96
97      CHK_SUCCESS_CLEANUP(
98                              clEnqueueNDRangeKernel(cmdQ_0, pKernel_s->kernel, 1, NULL,
                                    &(globalWorkSize[0]), &(localWorkSize[0]), 1, &
                                    firstRoundEvt, NULL); ,
99                              clReleaseMemObject(outputBufferD);
100     )
101
102     //******copy the resulting scalar into the user-provided output Buffer
103     //the value to be copied is outputBufferD[0]
104     CHK_SUCCESS_CLEANUP(copyBuffer(outputBufferD, *pOutputBuffer, sizeof(double)); ,
            clReleaseMemObject(outputBufferD);)
105
106
107     clReleaseMemObject(outputBufferD);
108
109     return ATOM(ok);
110
111 }
112
113
114 static ERL_NIF_TERM reduceDD(ErlNifEnv * env, int argc, const ERL_NIF_TERM argv[]) {
115
116     // %%reduceDD(Kernel::kernel(), InputBuffer::deviceBuffer(), OutputBuffer::
            deviceBuffer())
117     OCL_INIT_CHECK()
118     NIF_ARITY_CHECK(3)
```

```
119     /*get the parameters (Kernel::kernel(), InputBuffer::deviceBuffer(),
            OutputBuffer::deviceBuffer())**********************/
120
121     kernel_sync* pKernel_s = NULL;
122     if (!enif_get_resource(env, argv[0], kernel_sync_rt, (void**) &pKernel_s)) {
123         cerr << "ERROR :: reduceDD: 1st parameter is not a kernel_sync" << endl;
124         return enif_make_badarg(env);
125     }
126
127     cl_mem* pInputBuffer = NULL;
128     if (!enif_get_resource(env, argv[1], deviceBuffer_rt, (void**) &pInputBuffer) ||
             *pInputBuffer == NULL) {
129         cerr << "ERROR :: reduceDD: 2nd parameter is not a device buffer" << endl;
130         return enif_make_badarg(env);
131     }
132
133     cl_mem* pOutputBuffer = NULL;
134     if (!enif_get_resource(env, argv[2], deviceBuffer_rt, (void**) &pOutputBuffer)
            || *pOutputBuffer == NULL) {
135         cerr << "ERROR :: reduceDD: 3nd parameter is not a device buffer" << endl;
136         return enif_make_badarg(env);
137     }
138     /*****************************************************************************
            */
139
140     size_t szInputBufferByte = 0;
141
142     CHK_SUCCESS(getBufferSizeByte(*pInputBuffer, &szInputBufferByte);)
143
144     uint uiN = szInputBufferByte / sizeof(double);
145
146     if(!isPow2(uiN)) { //problem size MUST BE A POWER OF 2
147         cerr << "ERROR :: reduceDD: problem size is not a power of two." << endl;
148         return enif_make_badarg(env);
149     }
150
151     return reduceDD_Impl(env,pKernel_s, pInputBuffer, pOutputBuffer);
152
153 }
154
155 static ERL_NIF_TERM reduceDL_Impl(ErlNifEnv * env, kernel_sync* pKernel_s, cl_mem*
        pInputBufferD, cl_mem* pOutputBuffer) {
156
157 #if defined(TIME) || defined(TIME_FUN)
158     timespec
159     fun_start,
160     fun_end;
```

```
161
162     GET_TIME((fun_start));
163 #endif
164
165 #ifdef TIME
166     timespec
167     fun_prologue_end,
168     run_end,
169     marsh_end;
170 #endif
171
172     (void) pOutputBuffer; //unused
173
174     std::vector<cl_mem> buffers;
175
176     size_t szInputBufferDByte = 0;
177
178     CHK_SUCCESS(getBufferSizeByte(*pInputBufferD, &szInputBufferDByte);)
179
180     //*******allocate device input and output buffers having the size of the input
            and output ones**************
181     cl_mem outputBufferD = NULL;
182
183     size_t szOutputBufferDByte = sizeof(double);
184
185     CHK_SUCCESS_CLEANUP(createBuffer(szOutputBufferDByte, CL_MEM_READ_WRITE, &
            outputBufferD); , releaseBuffers(buffers); )
186     buffers.push_back(outputBufferD);
187
188 #ifdef TIME
189     GET_TIME((fun_prologue_end));
190 #endif
191
192     ERL_NIF_TERM returnTerm =
193     reduceDD_Impl( env, pKernel_s, pInputBufferD, &outputBufferD);
194
195 #ifdef TIME
196     GET_TIME((run_end));
197 #endif
198
199     if(returnTerm == ATOM(ok)) { // no errors occurred, return the resulting double
200
201         double *pBuffer = NULL;
202         CHK_SUCCESS_CLEANUP(mapBufferBlocking(outputBufferD, 0, szOutputBufferDByte,
                CL_MAP_READ, &pBuffer); , releaseBuffers(buffers);)
203
204         returnTerm = enif_make_list1(env, enif_make_double(env, pBuffer[0]));
```

```
205
206         unMapBuffer(outputBufferD, pBuffer);
207     }
208
209 #ifdef TIME
210     GET_TIME((marsh_end));
211 #endif
212
213
214     releaseBuffers(buffers);
215
216     ERL_NIF_TERM toReturn =  enif_make_tuple2(env, ATOM(ok), returnTerm);
217
218 #if defined(TIME) || defined(TIME_FUN)
219     GET_TIME((fun_end));
220 #endif
221
222 #if defined(TIME) || defined(TIME_FUN)
223
224     cerr <<
225     endl <<
226     "------------------------------------ ReduceDL (usec)
            ----------------------------------------" <<
227     endl <<
228     "Total run time: " << DIFF(fun_start, fun_end) << endl;
229 #endif
230
231 #ifdef TIME
232     cerr <<
233     "Function prologue time: " << DIFF(fun_start, fun_prologue_end) <<
234     endl <<
235     "KERNEL computation time: " << DIFF(fun_prologue_end, run_end) <<
236     endl <<
237     "Marshalling time: " << DIFF(run_end, marsh_end) <<
238     endl
239     ;
240 #endif
241
242     return toReturn;
243
244 }
245
246 static ERL_NIF_TERM reduceDL(ErlNifEnv * env, int argc, const ERL_NIF_TERM argv[]) {
247
248     // %%reduceHH(Kernel::kernel(), InputBuffer::deviceBuffer())
249     OCL_INIT_CHECK()
250     NIF_ARITY_CHECK(2)
```

```
251     /*get the parameters (Kernel::kernel(), InputBuffer::deviceBuffer())
            **********************/
252
253     kernel_sync* pKernel_s = NULL;
254     if (!enif_get_resource(env, argv[0], kernel_sync_rt, (void**) &pKernel_s)) {
255         cerr << "ERROR :: reduceDL: 1st parameter is not a kernel_sync" << endl ;
256         return enif_make_badarg(env);
257     }
258
259     cl_mem* pInputBufferD = NULL;
260     if (!enif_get_resource(env, argv[1], deviceBuffer_rt, (void**) &pInputBufferD)
            || *pInputBufferD == NULL) {
261         cerr << "ERROR :: reduceDL: 2nd parameter is not a host buffer" << endl ;
262         return enif_make_badarg(env);
263     }
264
265     /*************************************************************************
            */
266
267     ERL_NIF_TERM result =
268     reduceDL_Impl(env, pKernel_s, pInputBufferD, NULL);
269
270     if( enif_is_list(env, result))
271         return enif_make_tuple2(env, ATOM(ok), result);
272     else return result;
273
274 }
275
276
277 static ERL_NIF_TERM skeletonlib(ErlNifEnv * env, int argc, const ERL_NIF_TERM argv
        []) {
278     return enif_make_string(env,"Skel OCL ready",ERL_NIF_LATIN1);
279 }
```

B.2.4 Utilities: `utils.cpp`

```
1  //A synchronization barrier implemented using erlang NIF threading facilities
2  class Barrier {
3
4  private:
5    uint N_THREADS;
6
7    uint waiting_threads_counts[2];
8    uint current_counter;
9
10   ErlNifMutex* mtx;
11   ErlNifCond* cond;
```

```
12
13 public:
14   explicit Barrier(uint n_Threads) {
15     N_THREADS = n_Threads;
16     waiting_threads_counts[0] = waiting_threads_counts[1] = 0;
17     current_counter = 0;
18
19     mtx = enif_mutex_create((char*) "map_load_thread_cond_mtx");
20     cond = enif_cond_create((char*) "map_load_thread_cond");
21
22     }
23   ~Barrier() {
24     enif_cond_destroy(cond);
25     enif_mutex_destroy(mtx);
26   }
27
28   void await(){
29
30     enif_mutex_lock(mtx);
31
32     uint local_counter = current_counter;
33
34     waiting_threads_counts[local_counter]++;
35
36
37     if(waiting_threads_counts[local_counter] < N_THREADS) {
38
39       while(waiting_threads_counts[local_counter] < N_THREADS) {
40
41         enif_cond_wait(cond, mtx);
42       }
43     }
44     else {
45
46       current_counter ^= 1;
47       waiting_threads_counts[current_counter] = 0;
48
49       enif_cond_broadcast(cond);
50     }
51
52     enif_mutex_unlock(mtx);
53   }
54
55 };
56
57 /*
58 * Erlang list unamarshaling function.
```

```
59 * lastListCell is a pointer to the last cell of the list:
60 * valid when arrayLen is less than list length list, NULL when the list is over.
61 */
62 uint list_to_double_arrayN(ErlNifEnv *env, ERL_NIF_TERM list, double* array, uint
       arrayLen, ERL_NIF_TERM* lastListCell) {
63
64   if(array == NULL)
65     return 0;
66
67   uint len = 0;
68
69   if(enif_is_list(env, list)) {
70
71     ERL_NIF_TERM curr_cell = list;
72
73     uint i = 0;
74     for(i = 0; enif_is_list(env, curr_cell) && i < arrayLen; i++) {
75
76       ERL_NIF_TERM hd, tl;
77       if(!enif_get_list_cell(env, curr_cell, &hd, &tl))
78         break;//list is empty
79
80       if(!(enif_get_double(env, hd, &(array[i]) ))) {
81         std::cerr << "DEBUG: list_to_double_arrayN - Error: attempt to read float
               from something else!" << std::endl;
82       }
83       curr_cell = tl;
84
85     }
86
87     if(lastListCell != NULL) //user requested the term denoting the last cell
88       *lastListCell = curr_cell;
89
90     return i;
91
92   } else { //term "list" is not a list
93
94     #ifdef DEBUG
95     std::cerr << "DEBUG: list_to_double_arrayN - Error: trying to convert a non list
            as a list"<< std::endl ;
96     #endif
97
98     return 0;
99   }
100 }
101
102 //synchronized version
```

```
103 uint sync_list_to_double_arrayN(ErlNifEnv *env, ErlNifMutex *mtx, ERL_NIF_TERM list,
        double* array, uint arrayLen, ERL_NIF_TERM* lastListCell) {
104
105   enif_mutex_lock(mtx);
106
107   uint result = list_to_double_arrayN(env, list, array, arrayLen, lastListCell);
108
109
110   enif_mutex_unlock(mtx);
111
112
113   return result;
114
115 }
116
117 //Unmarshaling function
118 void list_to_double_array(ErlNifEnv *env, ERL_NIF_TERM list, double* array, uint
        listlen) {
119
120   if(array == NULL)
121     return;
122
123   if(enif_is_list(env, list)) {
124
125     list_to_double_arrayN(env, list, array, listlen, NULL);
126   }
127
128 }
129
130 //Marshaling function
131 ERL_NIF_TERM double_array_to_list(ErlNifEnv *env, double* array, size_t array_size)
        {
132
133   if(array == NULL)
134     return ATOM(error);
135
136   ERL_NIF_TERM* floatTermArray = (ERL_NIF_TERM*) enif_alloc(sizeof(ERL_NIF_TERM) *
          array_size);
137
138   for(uint i = 0; i < array_size; i++)
139     floatTermArray[i] = enif_make_double(env, array[i]);
140
141   ERL_NIF_TERM res = enif_make_list_from_array(env, floatTermArray, array_size);
142
143   enif_free(floatTermArray);
144
145   return res;
```

```
146 }
147
148 /*returns a string containing the content of the file at filePath
149  * otherwise "NULL".*/
150 string readFromFileStr(char* filePath) {
151
152   ifstream file(filePath, ios::in);
153   if (!file.is_open()) {
154     cerr << "Failed to open file for reading: " << filePath << endl;
155     return "NULL";
156   }
157
158   ostringstream oss;
159   oss << file.rdbuf();
160
161   return oss.str();
162 }
163
164 /*returns a dinamically allocated string containing the content of the file at
      filePath
165  * otherwise NULL.
166  * The returned sting must be enif_free'd when not needed.
167  * */
168 char* readFromFile(char* filePath) {
169
170   string srcStdStr = readFromFileStr(filePath);
171
172   char* toReturn = (char*) enif_alloc(srcStdStr.length()+1);
173
174   return strcpy(toReturn, srcStdStr.c_str());
175
176 }
177
178
179 #define MIN(a, b) ((a < b) ? a : b)
180
181 int inline isPow2(unsigned int v) { return v && !(v & (v - 1));}
182
183 /*
184  * A helper to return a value that is nearest value that is power of 2.
185  */
186 unsigned int nextPow2( unsigned int x )
187 {
188     --x;
189     x |= x >> 1;
190     x |= x >> 2;
191     x |= x >> 4;
```

```
192     x |= x >> 8;
193     x |= x >> 16;
194     return x+1;
195 }
196 
197 // Round Up Division function
198 size_t roundUp(uint group_size, uint global_size)
199 {
200     uint r = global_size % group_size;
201     if(r == 0)
202         return global_size;
203     else
204         return global_size + group_size - r;
205 }
206 
207 /*
208  * Compute the number of threads and blocks to use for the REDUCTION kernel.
209  * We set threads / block to the minimum of maxThreads and n/2 where n is
210  * problem size. We observe the maximum specified number of blocks, because
211  * each kernel thread can process more than 1 elements.
212  */
213 void getNumBlocksAndThreads(int n, int maxBlocks, int maxThreads, long unsigned int
        &blocks, long unsigned int &threads)
214 {
215         threads = (n < maxThreads*2) ? nextPow2((n + 1)/ 2) : maxThreads;
216         blocks = (n + (threads * 2 - 1)) / (threads * 2);
217 
218         if(maxBlocks > 0)
219           blocks = MIN(maxBlocks, blocks);
220 }
221 
222 /*
223  *  It finds all instances of a string in another string and replaces it with
224  *  a third string.
225  */
226 void replaceTextInString(std::string& text, std::string find, std::string replace)
227 {
228         std::string::size_type pos=0;
229         while((pos = text.find(find, pos)) != std::string::npos)
230         {
231             text.erase(pos, find.length());
232             text.insert(pos, replace);
233             pos+=replace.length();
234         }
235 }
```

B.2.5 OpenCL Kernels

Map Kernel

```
1  //Map Kernels, FUN_NAME is replaced at runtime with user-defined fuction's name;
       TYPE with double
2
3  //Unary Map kernel
4  __kernel void MapKernel(__global TYPE* input, __global TYPE* output, unsigned int
       outputOffset, unsigned int uiNumElements)
5  {
6      __private size_t i = get_global_id(0);
7
8    if(i >= uiNumElements)
9          return;
10
11   output[i + outputOffset] = FUN_NAME(input[i]);
12 }
13
14
15 //Binary Map kernel
16 __kernel void MapKernel2(__global TYPE* input1, __global TYPE* input2, __global TYPE
       * output, unsigned int outputOffset, unsigned int uiNumElements)
17 { __private size_t i = get_global_id(0);
18   if(i >= uiNumElements) return;
19   output[i + outputOffset] = FUN_NAME(input1[i], input2[i]);
20 }
```

Reduce Kernel

```
1  //Reduce kernel based on NVIDIA reduce6 kernel example
2  __kernel void ReduceKernel(__global TYPE* input, __global TYPE* output, unsigned int
       n, __local TYPE* localData) {
3    unsigned int blockSize = get_local_size(0);
4    unsigned int tid = get_local_id(0);
5    unsigned int i = get_global_id(0);
6    unsigned int gridSize = blockSize * get_num_groups(0);
7
8    TYPE result = 0;
9    if(i < n) {
10       result = input[i];
11       i += gridSize;
12   }
13   while(i < n) {
14       result = FUN_NAME(result, input[i]);
15       i += gridSize;
16   }
17
```

```
18    localData[tid] = result;
19
20    barrier(CLK_LOCAL_MEM_FENCE);
21
22    //unrolled loop
23     if(blockSize >= 512) {
24         if (tid < 256 && tid + 256 < n)
25             localData[tid] = FUN_NAME(localData[tid],localData[tid + 256]);
26         barrier(CLK_LOCAL_MEM_FENCE);
27     }
28     if(blockSize >= 256) {
29         if (tid < 128 && tid + 128 < n)
30              localData[tid] = FUN_NAME(localData[tid],localData[tid + 128]);
31         barrier(CLK_LOCAL_MEM_FENCE);
32     }
33     if(blockSize >= 128) {
34         if (tid <  64 && tid + 64 < n)
35             localData[tid] = FUN_NAME(localData[tid], localData[tid +  64]);
36         barrier(CLK_LOCAL_MEM_FENCE); }
37
38     if(blockSize >= 64) {
39         if (tid < 32 && tid + 32 < n)
40             localData[tid] = FUN_NAME(localData[tid], localData[tid +  32]);
41         barrier(CLK_LOCAL_MEM_FENCE);
42     }
43     if(blockSize >=  32) {
44         if (tid < 16 && tid + 16 < n)
45             localData[tid] = FUN_NAME(localData[tid], localData[tid +  16]);
46         barrier(CLK_LOCAL_MEM_FENCE);
47     }
48     if(blockSize >=  16) {
49         if (tid < 8 && tid + 8 < n)
50             localData[tid] = FUN_NAME(localData[tid], localData[tid +   8]);
51         barrier(CLK_LOCAL_MEM_FENCE);
52     }
53     if(blockSize >= 8) {
54         if (tid < 4 && tid + 4 < n)
55             localData[tid] = FUN_NAME(localData[tid], localData[tid +   4]);
56         barrier(CLK_LOCAL_MEM_FENCE);
57     }
58     if(blockSize >= 4) {
59         if (tid < 2 && tid + 2 < n)
60             localData[tid] = FUN_NAME(localData[tid], localData[tid +   2]);
61         barrier(CLK_LOCAL_MEM_FENCE);
62     }
63     if(blockSize >= 2) {
64         if (tid < 1 && tid + 1 < n)
```

```
65              localData[tid] = FUN_NAME(localData[tid], localData[tid +    1]);
66          barrier(CLK_LOCAL_MEM_FENCE);
67      }
68
69      if(tid == 0)  output[get_group_id(0)] = localData[tid];
70  }
```