REliable Power and time-ConstraInts-aware Predictive management of heterogeneous Exascale systems

RECIPE

WP2 Runtime Resource Management Infrastructure

D2.4 RECIPE Application Programming Interface

http://www.recipe-project.eu

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 801137

Grant Agreement No.: 801137
Deliverable: D2.4 RECIPE Application Programming Interface

Project Start Date: 01/05/2018 **Duration**: 36 months
Coordinator: *Politecnico di Milano, Italy*

Deliverable No:	D2.4
WP No:	2
WP Leader:	Giuseppe Massari
Due date:	31/10/2019
Delivery date:	31/10/2019

Dissemination Level:

PU	Public Use	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

DOCUMENT SUMMARY INFORMATION

Project title:	**REliable Power and time-ConstraInts-aware Predictive management of heterogeneous Exascale systems**
Short project name:	RECIPE
Project No:	801137
Call Identifier:	H2020-FETHPC-2017
Thematic Priority:	Future and Emerging Technologies
Type of Action:	Research and Innovation Action
Start date of the project:	01/05/2018
Duration of the project:	36 months
Project website:	http://www.recipe-project.eu

D2.4 RECIPE Application Programming Interface

Work Package:	WP2 Runtime Resource Management Infrastructure
Deliverable number:	D2.4
Deliverable title:	RECIPE Application Programming Interface
Due date:	31/10/2019
Actual submission date:	31/10/2019
Editor:	G. Agosta
Authors:	G. Agosta, A. Cilardo, G. Massari, F. Reghenzani
Dissemination Level:	PU
No. pages:	57
Authorized (date):	31/10/2019
Responsible person:	W. Fornaciari
Status:	Plan Draft **Working** Final Submitted Approved

Revision history:

Version	Date	Author	Comment
v.0.1	16/10/2019	G. Agosta	First draft
v.0.2	19/10/2019	G. Agosta	Python API for DSL
v.0.3	23/10/2019	A. Cilardo	Heterogeneous part added
v.0.4	24/10/2019	G. Massari	Runtime management API
v.0.5	24/10/2019	G. Agosta	Executive summary
v.1.0	31/10/2019	G. Massari	Overall revision after BSC review
v.1.1	02/11/2019	G. Massari	Final improvements

Quality Control:

	Who	Date
Checked by internal reviewer	BSC	30/10/2019
Checked by WP Leader	Giuseppe Massari	31/10/2019
Checked by Project Technical Manager	G. Agosta	31/10/2019
Checked by Project Coordinator	W. Fornaciari	31/10/2019

COPYRIGHT

ACKNOWLEDGEMENTS

RECIPE is a project that has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement No 801137. Please see http://www.recipe-project.eu for more information.

The partners in the project are Politecnico di Milano (POLIMI), Universitat Politècnica de València (UPV), Centro Regionale Information Communication Technology scrl (CeRICT), École Polytechnique Fédèrale de Lausanne (EPFL), Barcelona Supercomputing Center (BSC), Poznan Supercomputing and Networking Center (PSNC), IBT Solutions S.r.l. (IBTS), Centre Hospitalier Universitaire Vaudois (CHUV). The content of this document is the result of extensive discussions within the RECIPE ©Consortium as a whole.

DISCLAIMER

Contents

1 Runtime Managed Programming Models **8**
1.1 Adaptive Execution Model (AEM) . . . 8
1.1.1 Application requirements . . . 11
1.1.2 Run-time negotiation . . . 12
1.2 MANGOLIBS . . . 13
1.2.1 Application Architecture . . . 14
1.2.2 C++ API . . . 15
1.2.3 C Language API . . . 21
1.2.4 Sample Application: GIF FIFO . . . 24
1.2.5 Application Requirements . . . 28
1.2.6 Emulation Layer . . . 31
1.3 RECIPE Application Requirements . . . 31

2 Domain Specific Languages for RECIPE **33**
2.1 Python MANGOLIBS API . . . 33
2.1.1 Example Application . . . 37
2.2 OpenCL MANGOLIBS API . . . 38
2.2.1 Example Application . . . 40
2.3 Dynamic Compilation of Kernels . . . 41

3 Heterogeneous Acceleration Programming **43**
3.1 Implementation of OpenCL for the nu+ GPU-like programmable core . . . 44
3.1.1 OpenCL model on nu+ . . . 44
3.1.2 Designing OpenCL Host-Runtime on nu+ . . . 47
3.1.3 Compilation Support to OpenCL C Kernels . . . 48
3.2 HLS-targeted OpenCL programming interface . . . 51
3.2.1 OpenCL kernel design flow for FPGAs . . . 52
3.2.2 Improving Performance of OpenCL kernels . . . 53

4 Conclusions **57**

Executive Summary

This deliverable reports the progress of Task2.3 concerning the application programming interface developed for the RECIPE project. RECIPE leverages the existing MANGO API (and its MANGOLIB implementation) as the baseline for programming the heterogeneous accelerators. However, we also offer a more lightweight alternative in the Adaptive Execution Model. We also extend the MANGO API with Python bindings to provide the basis for the exploration of Domain Specific Language (DSL) features. Finally, for low-level accelerator programming, we provide OpenCL support to program the configurable GPU-like softcore *nu+*, which is part of the MANGO hardware unit portfolio, and evaluate the use of OpenCL-based HLS to generate custom accelerators.

RECIPE Technology	Baseline	Advance	Status
Runtime management interface	MANGOLIBS (C/C++ API)	Adaptive Execution Model (C/C++/Python)	Available
Domain Specific Language feature exploration	MANGOLIBS (C/C++ API)	Python API extended with dynamic recipe manipulation	Baseline Python API defined and implemented
Kernel compilation	Static kernel compilation only	Dynamic kernel compilation	Work in progress
Programming support for nu+	C only	OpenCL	Work in progress

1 Runtime Managed Programming Models

As already introduced in D2.1, for an effective management of HPC workload we think that a suitable integration between programming models and run-time management layers can play a key role. In this section, we provide an overview of the programming models we aim at using, for the implementation of the application use cases the RECIPE project is targeting. We introduce the *Abstract Execution Model* and the *MANGO API*. Both the programming models come from the development effort carried out during previous EU funded projects (FP7 2PARMA, FP7 HARPA and H2020 MANGO). They are characterized by a common feature: the integration of the application execution and lifecycle with the resource management actions.

Moreover, it is worth to specify that both the programming models are intended for parallel applications, running on a single local node of the HPC system. The usual *Message Passing Interface (MPI)* can be then exploited as complementary and additional choice, in order to scaling the performance, by distributing data and computation effort over multiple nodes of the HPC infrastructure.

1.1 Adaptive Execution Model (AEM)

This is the first programming model on which the BarbequeRTRM was based on [3]. It has been proposed for the implementation of *run-time adaptive* applications classes. This AEM comes with the resource management framework described later (the Barbeque Run-Time Resource Manager) and it is provided via the so called *Application RunTime Library (RTLib)*. The library includes functions and C++ class to enable the interaction between the application and the resource manager.

From the implementation standpoint, the *Adaptive Execution Model (AEM)* consists of implementing a C++ class, derived from the base class `BbqueEXC`, and include an instance of such a class in the application code, as shown in the example in Listing 1.

Listing 1: Definition of a BbqueEXC derived class

```
#include "bbque/bbque_exc.h"
#include "bbque/rtlib.h"

class  MyAppEXC: public BbqueEXC {
public:
        MyAppEXC(
                std::string const & name,
                std::string const & recipe,
                RTLIB_Services_t * rtlib);
        ~MyAppEXC();

private:
        RTLIB_ExitCode_t onSetup();
        RTLIB_ExitCode_t onConfigure(int8_t awm_id);
        RTLIB_ExitCode_t onRun();
        RTLIB_ExitCode_t onMonitor();
        RTLIB_ExitCode_t onSuspend();
        RTLIB_ExitCode_t onRelease();
};
```

The class represents an *Execution Context*, and is characterized by specific member functions to override. These functions, also shown in the boxes in Figure 1, are characterized by an `on-` prefix in the name – an approach similar to the one already known for programming Android systems. The implementation must in fact follow a given semantic, since the function execution is what we expect the application performs when entering a specific state. We can summarize the semantic of each member function as follows:

onSetup: Initialization code.

onConfigure: Called when the resource allocation changes. The application can configure itself according to the set of resources actually assigned.

onRun: The core of the algorithm. The part of the application performing "useful" computation.

onMonitor: Called after each `onRun` execution to check the current performance and send a feedback to the resource manager.

onRelease: Explicit deallocations (e.g. dynamic memory objects) and release.

Listing 2: Example of C++ main() function of an integrated application. The `MyAppEXC` class is derived from BbqueEXC.

```
 1  int main(int argc, char *argv[])
 2  {
 3      RTLIB_Services_t * rtlib;
 4      RTLIB_Init("MyApp", &rtlib);
 5
 6      MyAppEXC * pexc = std::make_shared<MyAppEXC>(
 7          "MyApp", "MyApp.recipe", rtlib);
 8      if (!pexc->isRegistered())
 9          return RTLIB_ERROR;
10
11      pexc->Start();
12      pexc->WaitCompletion();
13
14      return EXIT_SUCCESS;
15  }
```

In Listing 2 we can seen a minimal main function of AEM integrated application.

The main function must typically initialize the library by calling the `RTLIB_Init()` function and then instantiate an object of derived class type. This automatically spawns a control thread, in charge of performing the member function calls, according to the status messages coming from the resource manager.

After initializing the library, the application can instantiate an *execution context*. To do that, in the example the object of class `MyAppEXC` is created. The class is derived from `BbqueEXC`. If everything goes fine (line 8), the managed execution can be started by calling the `Start()`

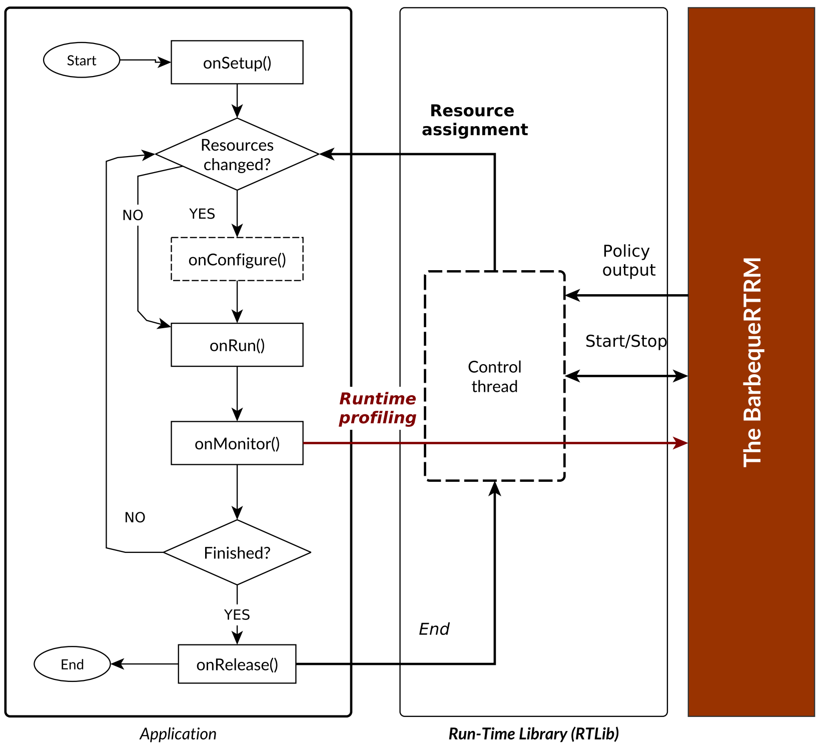

Figure 1: Adaptive Execution Model of BarbequeRTRM managed applications.

member function. This spawns the RTLib control thread and notifies the BarbequeRTRM of the application start, while the `main()` thread function will wait for the end of the managed execution (`WaitCompletion()`).

Therefore, on the derived class side, the AEM control thread will invoke the `onSetup` function for initialization purpose, before waiting for the BarbequeRTRM to assign resources. When this happens the `onConfigure` is called, so that the application can adapt its core algorithm to the actual availability of assigned system resources. Here some application-specific tuning actions can be performed.

Once the application is configured, the control thread can start a loop, in which member functions `onRun` and `onMonitor` are called in sequence. The loop goes on until the overall resource allocation remains unchanged. When the BarbequeRTRM performs a new run of the resource management policy, and this determines changes in the allocation of resources, the RTLib control thread is notified, such that the `onConfigure` function is called again. Then the regular execution of the application can be resumed.

The application terminates when a given exit condition is verified. In such a case, a specific exit code is returned by the `onRun` function, to notify the control thread and jump into the `onRelease` before terminating.

Application requirements

The current version of the BarbequeRTRM allows the application developer to suggest to the resource manager the optimal set of resource assignment configurations [5]. We refer to these configurations by using the expression *Application Working Modes* (AWM). For each working mode we typically have two mandatory pieces of information to specify:

1. The required hardware resources;
2. The expected performance level due to the assignment of the required resources (*value*).

From the application side, in fact, the outcome of a BarbequeRTRM resource allocation policy generally consists of the assignment of an AWM, with the consequent mapping of the resource requests onto the available system resources. Depending on the specific policy implementation, the assigned AWM can be built on-the-fly at runtime or it can be selected from a list provided by the developer. The latter case requires the developer to include this list in what is called the application *Recipe*, i.e. an XML file similar to Listing 3.

Listing 3: Example of application Recipe.

```xml
<?xml version="1.0"?>
<BarbequeRTRM recipe_version="0.8">
   <application priority="4">
      <platform id="org.linux.cgroup" hw="mango">
         <awms>
            <awm id="0" value="1" config-time="150">
               <resources>
                  <cpu>
                     <pe qty="100"/>
                     <mem qty="2" units="MB" />
                  </cpu>
                  <net qty="50" units="Kbps">
               </resources>
            </awm>
            ...
            <awm id="2" value="4" config-time="150">
               <resources>
                  <cpu>
                     <pe qty="200"/>
                     <mem qty="10" units="MB"/>
                  </cpu>
                  <acc>
                     <pe qty="4"/>
                  </acc>
                  <net qty="100" units="Kbps">
               </resources>
            </awm>
         </awms>
      </platform>
   </application>
</BarbequeRTRM>
```

The file is usually located under a specific installation sub-directory of the BarbequeRTRM framework (`$(ROOT_DIR)/etc/bbque/recipes/`) and must necessarily end with `.recipe` extension. To specify the recipe file associated to the starting execution context, the second argument of the `BbqueEXC` constructor is used, as shown in Listing 2. In the example provided in Listing 3, we assigned to the application a static *priority* level equal to 4, (where 0 is the highest priority value we can assign). This is an information that resource allocation policy should typically take into account.

The `platform` section then identifies the target system we are referring to. In this case, a system running a Linux OS (`id=‘‘org.linux.cgroup’’`) whose architecture is better qualified by the attribute `hw=‘‘mango’’`. The set of Application Working Modes (`amws`) is thus platform-specific. Therefore the Recipe format allows the developer to include several platform sections into a single Recipe file. The proper section will be parsed at the application start-time depending on the actual system.

Looking at each Application Working Mode definition, we can specify several attributes: 1) the progressive numeric identifier (`id`); 2) an optional descriptive name; 3) the aforementioned performance `value`; 4) the configuration time profiled, in milliseconds (`config-time`). Specifically, the last two attributes provide input data to the resource allocation policy. The *value* is indeed a number expressing the level of performance or QoS, while the `config-time` keeps track of the time overhead experienced to adapt the application to the related AWM assignment.

The `<resources>` subsections then, make up the part of the Recipe in charge of reporting the resource requirements of each AWM. In the example, such requirements are expressed in terms of CPU bandwidth time quota (`pe` under `cpu`), amount of memory (`mem`), number of accelerator cores (`pe` under `acc`), and network bandwidth (`net`). Concerning the specific case of the CPU time, the values reported must be read as percentages. Therefore, values greater than 100 simply expresses the usage requirement of more than one CPU core. Generally, a good practice would be to write an application Recipe through a suitable profiling of the application execution, under different resource assignment configurations.

Run-time negotiation

We shown how the AEM drives the execution of applications by invoking `onSetup`, `onConfigure`, `onRun`, `onMonitor`, `onSuspend`, and `onRelease` callbacks, following the proposed execution flow. This run-time managed execution of the application allows us to perform some kind of performance monitoring at RTLib level.

In this regard, the library exposes some functions through which the application can:

- Be aware of the current resource allocation (*resource awareness*)
- Express its Quality of Service goals (*runtime profiling*)
- Assess its current performance (*performance awareness*)

Resource awareness. Applications can be aware of the amount of currently allocated resources. These APIs are usually used by applications in the `onConfigure` method. However, they can be anywhere in the application code. Allocation data is stored locally to the RTLib. It is updated each time the resource manager changes the resource allocation.

`RTLIB_ExitCode_t GetAssignedResources (RTLIB_ResourceType_t r_type, int32_t & r_amount);`
Get the amount of assigned resources of type *r_type*. Available resource types are:
`RTLIB_ResourceType::CPU`: number of cores
`RTLIB_ResourceType::PROC_NR`: number of processors
`RTLIB_ResourceType::PROC_ELEMENT`: CPU bandwidth
`RTLIB_ResourceType::MEMORY`: amount of memory
`RTLIB_ResourceType::GPU`: amount of GPUs
`RTLIB_ResourceType::ACCELERATOR`: amount of cores in accelerators
The amount of resources is written in *r_amount*.

Runtime Profiling. Applications can declare their performance goal in terms of CPS goal, i.e., throughput. CPS (Cycles Per Second) is the number of processing cycles (i.e. `onRun` invocations) completed in a second. The goal value is not forwarded immediately to the resource manager. Conversely, it is locally used by the RTLib, which constantly monitors the current performance and notifies the resource manager only when the measured CPS does not match the declared CPS goal, thus minimizing communication overheads. The current distance between instantaneous and goal CPS is defined as *goal gap*. It is computed as follows:

$$goal\ gap = \frac{CPS_{current} - CPS_{goal}}{CPS_{goal}} \tag{1}$$

For example, an application whose goal is 2.0 CPS and whose current CPS is 1.8 has a goal gap of $\frac{1.8-2-0}{2.0} = -0.10$, i.e. is 10% too slow.

`RTLIB_ExitCode_t SetCPSGoal (float cps_min);`
 `RTLIB_ExitCode_t SetCPSGoal (float cps_min, float cps_max);`

Set the desired performance in terms of Cycles Per Second (CPS). The resource manager will be automatically notified when the instantaneous CPS will be lower than `cps_min` or, if declared, higher that `cps_max`.
RTLIB_ExitCode_t SetGoalGap(int goal_gap);
Explicitly set a Goal Gap to the resource manager. The `goal_gap` must be computed using Equation 1.
RTLIB_ExitCode_t SetCPS(float cps);
Force maximum CPS to be `cps`. If the application is faster than that, maximum CPS is enforced by the RTLib using sleep. If used along with the `SetCPSGoal` function, after the sleep the resource manager is notified that the application is too fast, and will reduce allocated resources accordingly.
RTLIB_ExitCode_t SetCTimeUs(uint32_t us);
The same as `SetCPS`, but with cycle time ($\frac{1}{CPS}$).

1.2 MANGOLIBS

MANGOLIBS is a runtime support library that allows programmers to develop parallel applications leveraging heterogeneous accelerators. Applications developed with MANGOLIBS do not need to explicitly manage the resources, which are instead managed by the Barbeque Runtime

Resource Manager, a state of the art open source resource manager. Thus, system level metrics such as resource utilisation and power can be optimised while preserving the quality of service requested by the application. Furthermore, MANGOLIBS can be easily extended to support new heterogeneous programmable accelerators, making it a tool for the exploration of heterogeneous architectures in the context of High Performance Computing (HPC).

Th MANGO programming model was defined and implemented by the MANGO consortium [4]. The definition is publicly available through the deliverables of the MANGO project [2, 1]. It is implemented as a set of open source libraries [1], natively defined in C++, but with C bindings available out of the box. We summarize here the application architecture for ease of reference.

Application Architecture

MANGOLIBS defines three main classes of objects that are meant to reside on the accelerator cluster, namely *kernels*, *buffers* and *events*.

Kernel objects represent a fragment of code, encapsulated as a C/C++ function, that will run on an accelerator. The kernel is defined in a separate compilation unit, using a `mango_kernel` keyword to denote the entry point of the compilation unit. Each kernel can receive as parameters either pointers to buffers or events, or scalar values (or a mix of both). Multiple kernels may be defined by an application to be run concurrently on different compute units. Each kernel may have multiple implementations targeting different types of compute units. Each kernel is associated with a completion event that can be waited upon from the host to suspend execution while waiting for the kernel to complete.

Buffer objects represent a region of memory that can be accessed by a given set of kernels. An associated event can be used to notify read or write actions on the buffer. Buffers are accessed directly (via their memory address in the compute unit's address space) from the accelerator, and via `write` and `read` primitives on the host side. Standard buffers perform host-accelerator data transfer synchronously, but FIFO buffers are also provided to perform asynchronous data transfer in bursts.

Events are mapped to special memory regions on the accelerator cluster that are accessed in mutual exclusion. They allow synchronization among concurrent kernels and with the host.

On the host side, MANGOLIBS also defines a *context* that allows to encapsulate the information about the application that need to be passed to the resource manager (in particular, the *recipe*, that is the set of configurations and requirements in terms of compute units, memory, and bandwidth, that the application needs to reach a specific operating point) and to ask for resource allocation and deallocation. The kernels, buffers and events used by an application are passed to the context in the form of a *task graph*, so that the resource manager can consider all the elements of the application in the frame of the flow of data among them.

This is the biggest difference with respect to programming models like OpenCL. The MANGOLIBS API in fact, simply requires that the developers would build a *task-graph* based description of the application, by using suitable library functions. An example of task-graph is shown in Figure 2. The nodes can represent a task or a memory buffer, properly placed for data exchanging.

[1]https://bitbucket.org/mango_developers/mangolibs

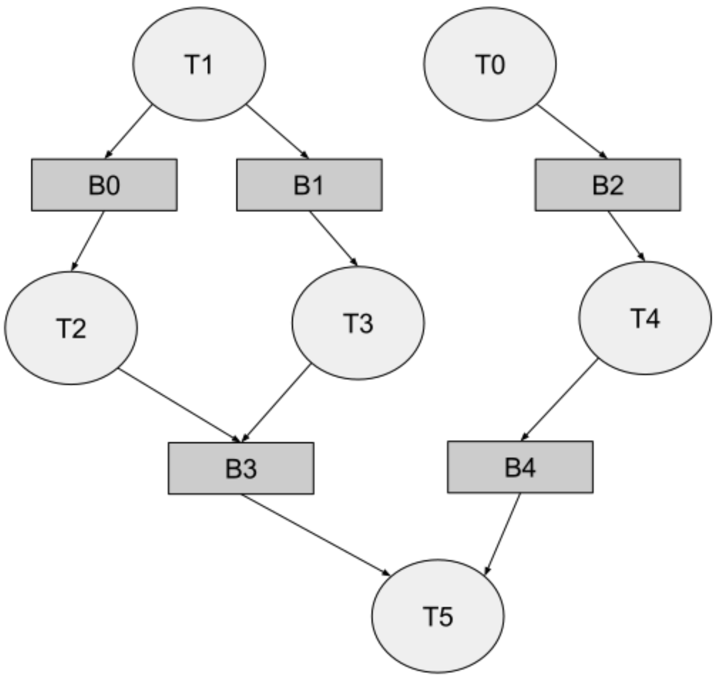

Figure 2: Task-graph based representation of an application.

Thanks to a tight integration of the MANGO runtime with the local resource manager (the BarbequeRTRM), the application developer is relieved from the burden of taking care of resource mapping. This means that the tasks will be offloaded to specific computing units, and the buffers will be allocated to memory nodes, in a transparent manner and according to the local resource manager policy.

One of the goals of this programming model is therefore to abstract both the local resource manager and the low-level access to the hardware. Therefore, the MANGOLIBS implementation relies on two internal APIs: one towards the hardware abstraction layer, and one towards the runtime resource manager. The LIBHN API allows the MANGOLIBS to interact with the allocated resources, by loading and running kernels as well as writing and reading from memory objects (buffers and events). The runtime library API allows MANGOLIBS to interact with the resource manager, performing allocation and deallocation requests, and notifying relevant events (e.g., kernel completion).

The current implementation of the programming model comes with the *MANGO library (libmango)* and provides to the developer a C and a C++ API, as described as it follows.

C++ API

The C++ API is the main implementation of the `libmango`. The library provides access to both the two low-level library implementations: one targeting the `hnlib`, described in D2.1, and one targeting an emulated accelerator device, which is in turn implemented as a set of Linux processes (one per emulated kernel).

The main class `mango::BBQContext` represents the interface with the underlying runtime framework. Any MANGO application must initialize one, and only one, object of this class to be recognized by the resource manager.

mango::BBQContext

- `BBQContext (std::string const &_name="app", std::string const &_recipe="generic")`
  - *Performs the necessary internal initialization of* `libmango` *and underlying libraries (including the connection with* `libhn`*).*
- `virtual mango_exit_code_t resource_allocation (TaskGraph &tg) noexcept override`
  - *Performs the resource allocation of the specified task graph of the application. This function is blocking until Barbeque assigns the resources.*
- `virtual mango_exit_code_t resource_deallocation (TaskGraph &tg) noexcept override`
  - *Deallocates the previous assigned resources.*
- `virtual std::shared_ptr< Event > start_kernel (std::shared_ptr< Kernel > kernel, KernelArguments &args, std::shared_ptr< Event > _e=nullptr) noexcept override`
  - *Launches a given kernel with the given parameters. This method must be called after a successful allocation of the resources. The event parameter can be used to retrieve the termination event of the kernel automatically created.*

An application is composed by a non-empty set of kernels. These kernels are represented with instances of the following class. Each kernel is then run, according to the resource manager decision, onto a computing resource (e.g., a HW accelerator) located on FPGA.

mango::Kernel

- `Kernel (mango_id_t kid, KernelFunction *k, std::vector< mango_id_t > buffers_in, std::vector< mango_id_t > buffers_out) noexcept`
  - *The class constructor. A kernel is defined over a numerical id, a KernelFunction and by the sets of input and output buffers.*
- `bool operator== (const Kernel &other) const noexcept`
- `bool is_a_reader (mango_id_t buffer_id) const noexcept`
  - *Return true if the kernel is a reader of the buffer provided as input. False otherwise.*
- `bool is_a_writer (mango_id_t buffer_id) const noexcept`
  - *Return true if the kernel is a writer of the buffer provided as input. False otherwise.*
- `std::shared_ptr< KernelCompletionEvent > get_termination_event () noexcept`
- `std::vector< std::shared_ptr< Event > > & get_task_events () noexcept`
- `const std::vector< std::shared_ptr< Event > > & get_task_events () const noexcept`
- `mango_addr_t get_virtual_address () const noexcept`
  - *Return the virtual address associated to the kernel binary image.*
- `mango_addr_t get_physical_address () const noexcept`
  - *Return the physical address associated to the kernel binary image.*
- `mango_id_t get_mem_tile () const noexcept`
  - *Return the memory node associated to the kernel binary image, if the target system included more than one memory on the platform.*
- `std::shared_ptr< Unit > get_assigned_unit () const noexcept`
  - *Return the type of processing unit assigned to the kernel.*
- `const KernelFunction * get_kernel () const noexcept`
  - *Return the associated kernel function.*
- `mango_id_t get_id () const noexcept`
- `std::vector< mango_id_t >::const_iterator buffers_in_cbegin () const noexcept`
- `std::vector< mango_id_t >::const_iterator buffers_in_cend () const noexcept`
- `std::vector< mango_id_t >::const_iterator buffers_out_cbegin () const noexcept`
- `std::vector< mango_id_t >::const_iterator buffers_out_cend () const noexcept`
- `void set_thread_count (int thread_count) noexcept`
  - *Set the number of threads of this kernel. This information will be provided to the resource manager.*
- `int get_thread_count () const noexcept`
  - *Get the number of threads of this kernel.*

Each kernel can write from or read to a buffer local to the processing unit. These buffers are memory portions usually shared among different cores of the processing unit and they are represented by instances of the following class:

mango::Buffer

- **Buffer** (mango_id_t bid, mango_size_t size, const std::vector< mango_id_t > &kernels_in={}, const std::vector< mango_id_t > &kernels_out={}) noexcept
- `virtual std::shared_ptr< const Event > write (const void *GN_buffer, mango_size_t global_size=0) const noexcept`
  - *Memory transfer from general-purpose node (GN) to the FPGA (heterogeneous node (HN)) in DIRECT mode.*
- `virtual std::shared_ptr< const Event > read (void *GN_buffer, mango_size_t global_size=0) const noexcept`
  - *Memory transfer from heterogeneous node (HN) to general-purpose node (GN) in DIRECT mode.*
- `bool isReadByHost ()`
  - *Check whether the buffer is read by the host.*
- `bool isReadBy (uint32_t kid) const noexcept`
  - *Check whether the buffer is read by a given kernel.*
- `bool operator== (const Buffer &other) const noexcept`
- `mango_exit_code_t resize (mango_size_t size) noexcept`
- `mango_id_t get_id () const noexcept`
  - *Get the event identifier.*
- `mango_id_t get_size () const noexcept`
  - *Get the size of th buffer.*
- `mango_size_t get_phy_addr () const noexcept`
  - *Get the physical address of the event. This is not an actual physical address, but it represents an offset in the tile register.*
- `mango_id_t get_mem_tile () const noexcept`
  - *Get the identifier of the memory tile assigned.*
- `void set_phy_addr (mango_size_t addr) noexcept`
  - *Get the physical address of the event. This is not an actual physical address, but it represents an offset in the tile register.*
- `void set_mem_tile (mango_id_t tile) noexcept`
  - *Get the identifier of the memory tile assigned.*
- `std::shared_ptr< Event > get_event () noexcept`
  - *It returns the pointer to the event.*
- `const std::vector< mango_id_t > & get_kernels_in () const noexcept`
- `const std::vector< mango_id_t > & get_kernels_out () const noexcept`

To build the kernel object the KernelFunction object must be also set. This class represents an available binary for a specific architecture for the kernel.

mango::KernelFunction

- `mango_exit_code_t load (const std::string &kernel_file, mango_unit_type_t unit, mango_file_type_t type) noexcept`
  - *Load a new kernel image.*
- `std::string get_kernel_version (mango_unit_type_t type) const`
  - *Given a Mango unit type, it returns the matching filename.*
- `void set_kernel_size (mango_unit_type_t type, mango_size_t size)`
- `mango_size_t get_kernel_size (mango_unit_type_t type) const`
  - *Given a Mango unit type, it returns the matching file size.*
- `std::map< mango_unit_type_t, mango_size_t >::const_iterator cbegin () const noexcept`
- `std::map< mango_unit_type_t, mango_size_t >::const_iterator cend () const noexcept`
- `bool is_loaded () const noexcept`
- `size_t length () const noexcept`

mango::Arg

- `mango_size_t get_value () const noexcept`
- `mango_size_t get_size () const noexcept`
- `mango_id_t get_id () const noexcept`
- `void set_value (mango_size_t value) noexcept`

An object instance of the Event instead represents a generic event. It can represent a kernel event, e.g. the end of a kernel execution, a buffer event, e.g. the finishing of a write or read of a shared memory area, or a custom event directly managed and triggered by the application. The first two are generated by `libmango` and obtained through appropriate function calls from classes Kernel and Buffer.

mango::Event

- `Event (mango_id_t kernel_id) noexcept`
- `Event (const std::vector< mango_id_t > &kernel_id_in, const std::vector< mango_id_t > &kernel_id_out) noexcept`
- `virtual void wait_state (uint32_t state) const noexcept`
- `virtual uint32_t wait () const noexcept`
  - *High level wait primitive.*
- `virtual void write (uint32_t value) const noexcept`
  - *Set an event value.*
- `uint32_t read () const noexcept`
  - *Read and reset an event read value from the TILEREG register associated with event and replace it with 0. This one should deprecated.*
- `bool operator== (const Event &other) const noexcept`
- `mango_id_t get_id () const noexcept`
  - *Get the event identifier.*
- `mango_size_t get_phy_addr () const noexcept`
  - *Get the physical address of the event. This is not an actual physical address, but it represents an offset in the tile register.*
- `void set_phy_addr (mango_size_t addr) noexcept`
  - *Get the physical address of the event. This is not an actual physical address, but it represents an offset in the tile register.*
- `void set_fifo_task (std::unique_ptr< std::thread > task) noexcept`
  - *Set a callback function for write/read data asynchronously.*
- `const std::vector< mango_id_t > & get_kernels_in () const noexcept`
- `const std::vector< mango_id_t > & get_kernels_out () const noexcept`
- `template<typename A , typename B > void set_callback (A _bbq_notify_callback, B obj, mango_id_t _id) noexcept`

Instances of Kernel, Buffer, and Event classes are then grouped into a task graph. This data structure is built using the appropriate class:

mango::TaskGraph

- `TaskGraph (std::initializer_list< std::shared_ptr< Kernel >> lkernels, std::initializer_list< std::shared_ptr< Buffer >> lbuffers, std::initializer_list< std::shared_ptr< Event >> levents={}) noexcept`
  - *Define a task graph.*
- `TaskGraph (int k, int b, int e,...)`
  - *Define a task graph.*
- `~TaskGraph ()`
  - *Destroy a task graph.*
- `TaskGraph & operator+= (std::shared_ptr< Kernel > kernel)`
  - *Add a kernel to the task graph.*
- `TaskGraph & operator-= (std::shared_ptr< Kernel > kernel)`
  - *Remove a kernel from the task graph.*
- `TaskGraph & operator+= (std::shared_ptr< Buffer > buffer)`
  - *Add a buffer to the task graph.*
- `TaskGraph & operator-= (std::shared_ptr< Buffer > buffer)`
  - *Remove a buffer from the task graph.*
- `TaskGraph & operator+= (std::shared_ptr< Event > event)`
  - *Add an event to the task graph.*
- `TaskGraph & operator-= (std::shared_ptr< Event > event)`
  - *Remove an event from the task graph.*
- `std::shared_ptr< Kernel > get_kernel_by_id (mango_id_t id) noexcept`
- `std::vector< std::shared_ptr< Kernel > > & get_kernels () noexcept`
- `std::vector< std::shared_ptr< Buffer > > & get_buffers () noexcept`
- `std::vector< std::shared_ptr< Event > > & get_events () noexcept`

C Language API

The C language API is a wrapper around the C++ API that is provided both for compatibility with C code and for compatibility with the early version of the library, which was developed in C.

All the data types in the function prototypes have been made opaque using specific `typedef` types. This hides to the application some specific type of the machine, such as the size of the memory addresses. In the current MANGO implementation, the used types are mostly `uint32_t`, due to the addressing size that is of 32-bit.

We grouped the API in 8 groups: initialization and shutdown, kernel loading, task graph definition, task graph registration, resource allocation, kernel launching, synchronization primitives, and data transfer.

The first group regards the initialization and the shutdown. As the name suggests, it basically wraps the BBQContext initialization functions.

Initialization and shutdown

- mango_exit_t mango_init (const char *application_name, const char *recipe)
  - *Initialize runtime library.*
- mango_exit_t mango_release ()
  - *Shutdown runtime library.*

The second logical step for the application is to provide the information about the binaries available for the kernels. The following functions are mainly a wrap of the KernelFunction class.

Kernel loading

- kernelfunction * mango_kernelfunction_init ()
  - *Initialize a kernel function data structure.*
- mango_exit_t mango_load_kernel (const char *kname, kernelfunction *kernel, mango_unit_type_t unit, filetype t)
  - *Load a kernel binary of a targer architetecture, ready for offloading.*

Then, the task graph has to be built. Thus all the `mango_register_*` allows the developer to create all the task graph components.

Registration of task graph components

- mango_kernel_t mango_register_kernel (uint32_t kernel_id, kernelfunction *kernel, unsigned int nbuffers_in, unsigned int nbuffers_out,...)
  - *Register a kernel to add to the task-graph.*
- void mango_deregister_kernel (mango_kernel_t kernel)
  - *De-register the kernel to remove from the task-graph.*
- mango_buffer_t mango_register_memory (uint32_t buffer_id, size_t size, mango_buffer_type_t mode, unsigned int nkernels_in, unsigned int nkernels_out,...)
  - *Register a memory region to use as a buffer for communication.*
- void mango_deregister_memory (mango_buffer_t mem)
  - *Deallocate registered memory.*
- mango_event_t mango_register_event (unsigned int nkernels_in, unsigned int nkernels_out,...)
  - *Register a synchronization event (semaphore?) that will be used on the heterogeneous node side (FPGA)*
- void mango_deregister_event (mango_event_t event)
  - *Deallocate registered event.*
- mango_event_t mango_get_buffer_event (mango_buffer_t buffer)
  - *Get an event from the corresponding buffer. This function is needed to keep mango_buffer_t opaque in the C interface.*

Once all the components of the task graph are created, it is necessary to specify the structure of the task graph, i.e. the relation between kernel, buffers, and events. This is done thanks to the following API.

Task graph definition

- mango_task_graph_t * mango_task_graph_vcreate (mango_kernel_t **kernels, mango_buffer_t **buffers, mango_event_t * * events)
  - *Define a task graph.*
- mango_task_graph_t * mango_task_graph_create (int k, int b, int e,...)
  - *Define a task graph.*
- void mango_task_graph_destroy (mango_task_graph_t *task_graph)
  - *Destroy a task graph.*
- void mango_task_graph_destroy_all (mango_task_graph_t *task_graph)
  - *Destroy a task graph and deregister all of its components.*
- mango_task_graph_t * mango_task_graph_add_kernel (mango_task_graph_t *tg, mango_kernel_t *kernel)
  - *Add a kernel to the task graph.*
- mango_task_graph_t * mango_task_graph_remove_kernel (mango_task_graph_t *tg, mango_kernel_t *kernel)
  - *Remove a kernel from the task graph.*
- mango_task_graph_t * mango_task_graph_add_buffer (mango_task_graph_t *tg, mango_buffer_t *buffer)
  - *Add a buffer to the task graph.*
- mango_task_graph_t * mango_task_graph_remove_buffer (mango_task_graph_t *tg, mango_buffer_t *buffer)
  - *Remove a buffer from the task graph.*
- mango_task_graph_t * mango_task_graph_add_event (mango_task_graph_t *tg, mango_event_t *event)
  - *Add an event to the task graph.*
- mango_task_graph_t * mango_task_graph_remove_event (mango_task_graph_t *tg, mango_event_t *event)
  - *Remove an event from the task graph.*

After the creation of the task graph, the allocation of the resources is requested to the resource manager, in our case BarbequeRTRM, using the following functions:

Resource Allocation

- mango_exit_t mango_resource_allocation (mango_task_graph_t *tg)
  - *Resource allocation for the task-graph.*
- void mango_resource_deallocation (mango_task_graph_t *tg)
  - *Resource de-allocation for the task-graph.*

At this point, the resources are allocated and the kernel execution can be started. This is done via the following functions:

Kernel launch

- mango_arg_t * mango_arg (mango_kernel_t kernel, const void *value, size_t size, mango_buffer_type_t t)
  - *Build an argument parameter.*
- mango_args_t * mango_set_args (mango_kernel_t kernel, int argc,...)
  - *Set up the arguments for a kernel.*
- mango_event_t mango_start_kernel (mango_kernel_t kernel, mango_args_t *args, mango_event_t event)
  - *Run a kernel.*

During the execution of the kernels, the following functions enable the use of synchronization registers, i.e. events from the application point of view. These functions enable the GN-side threads to synchronize with the kernels running on the HN.

Synchronization primitives

- void mango_wait (mango_event_t e)
  - *High level wait primitive.*
- void mango_wait_state (mango_event_t e, uint32_t state)
  - *High level wait primitive.*
- void mango_write_synchronization (mango_event_t event, uint32_t value)
  - *Initialize an event.*
- uint32_t mango_read_synchronization (mango_event_t event)
  - *Read and reset an event.*
- uint32_t mango_lock (mango_event_t e)
  - *Lock and read an event*

Finally, the following functions provide the option to exchange data from the GN to the HN and vice versa.

GN-HN data transfer

- mango_event_t mango_write (const void *GN_buffer, mango_buffer_t HN_buffer, mango_communication_mode_t mode, size_t global_size)
  - *Memory transfer from GN to HN.*
- mango_event_t mango_read (void *GN_buffer, mango_buffer_t HN_buffer, mango_communication_mode_t mode, size_t global_size)
  - *Memory transfer from HN to GN.*

Sample Application: GIF FIFO

In this subsection, we discuss the sample application "GIF FIFO" included in the release of the MANGO software[2]. The application performs a simple scaling of an input image in GIF format. Listing 4 shows the usage of the C++ version of the MANGO API. It represents the part of the application running on a general-purpose node, i.e. CPU, and thus linked to the MANGO programming model library.

[2]https://bitbucket.org/mango_developers/mangolibs

As previously introduced, a mandatory initial step of the application execution is the initialization of the context. This is performed at line 9, by instancing the object `BBQContext`, providing the name of the application and the recipe file as arguments (see Section 1.2.5).

Listing 4: main() function of the GIF_FIFO sample application. This is the host-side of the application, running on the CPUs of the general-purpose node (GN)

```
 1
 2 int main () {
 3     // Load input image...
 4     // ...
 5     int SX=512;
 6     int SY=512;
 7
 8     // Initialization
 9     context = new mango::BBQContext("gif_animation", "gif_animation");
10
11     // Kernel objects
12     auto kf_scale = new mango::KernelFunction();
13     kf_scale->load(kernel_binary_path_cpu,
14                    mango::UnitType::GN, mango::FileType::BINARY);
15     kf_scale->load(kernel_binary_path_peak,
16                    mango::UnitType::PEAK, mango::FileType::BINARY);
17     // ...
18
19     // Task graph construction
20     auto kscale  = context->register_kernel(KSCALE, kf_scale, {B1}, {B2});
21     auto b1 = context->register_buffer(
22         B1, SX*SY*3*sizeof(Byte), {}, {KSCALE}, mango::BufferType::FIFO);
23     auto b2 = context->register_buffer(
24         B2, SX*2*SY*2*3*sizeof(Byte), {KSCALE}, {}, mango::BufferType::
               FIFO);
25
26     // Resource allocation
27     tg = new mango::TaskGraph({ kscale }, { b1, b2 });
28     context->resource_allocation(*tg);
29
30     // Execution setup
31     auto argB1 = new mango::BufferArg( b1 );
32     auto argB2 = new mango::BufferArg( b2 );
33     auto argSX = new mango::ScalarArg<int>( SX );
34     auto argSY = new mango::ScalarArg<int>( SY );
35     auto argE1 = new mango::EventArg( b1->get_event() );
36     auto argE2 = new mango::EventArg( b2->get_event() );
37     argsKSCALE = new mango::KernelArguments(
38         { argB2, argB1, argSX, argSY, argE1, argE2 }, kscale);
39
40     // FIFO data transfer
41     b1->write(in, 4*SX*SY*3*sizeof(Byte));
42     b2->read(out, 4*SX*2*SY*2*3*sizeof(Byte));
43
44     // Kernel execution
45     auto e3 = context->start_kernel(kscale, *argsKSCALE);
46     e3->wait();
47
48     // Deallocation and tear-down
49     context->resource_deallocation(*tg);
50      //
51      // Save output image...
52      // ...
53      return 0;
54  }
```

We can then proceed with the instantiation of a kernel object for each application task (line 11). In this case, the application includes only the task to perform the scaling of the input image (`kf_scale`). The object is hence filled with references to the executable binaries of all the architectures for which it has been pre-compiled (in this case, CPU and PEAK).

The next step is, the construction of the task-graph representation. We start from registering the kernels (line 19), by specifying an integer id (macro `KSCALE`), the kernel object, and the lists of input and output buffers (ids). Similarly, the registration of the buffers, with the size, kernels reading from and writing to, the id number (macros `B1` and `B2`), the buffer type (`FIFO` or `DIRECT`).

The `TaskGraph` object is then instantiated by providing the list of kernels and buffers registered above. At this point, everything is ready for the resource allocation request. The task-graph is sent via the `resource_allocation()` function of the context object (line 28). When the function returns, a set of resources is available and the application is almost ready to start the useful execution, i.e. the kernels.

In fact, before proceeding with the kernel launch, we may need to properly set the arguments. Lines 31-38 show the different classes of arguments currently supported. These are packed into a single `KernelArgument` object, which will be passed to the `start_kernel()` context member function. Input data (`in`) are written in the input buffer (`b1`), while we setup the read operation from the output buffer (`b2`).

The kernel execution can be launched with the `start_kernel()` which transfers the correct executable to the assigned processing unit of the HN, along with its arguments, before starting it. The call returns an `Event` object (`e3`) that can be used for synchronizing the main thread running on the GN, with the termination of the kernel running on a HN unit (line 46).

Finally, resources are released when `resource_deallocation()` is called.

Listing 5 instead, shows the code that is compiled to run on a HN unit, and sent by the `start_kernel()` call. The entry point is the `main` function (line 7), which first initializes the environment by calling the `mango_init()` function. This means for instance that some default event resources are allocated and initialized (e.g. start/stop of the kernel, barrier for multi-threading, etc. . .).

The `mango_memory_map()` calls instead, are needed in order to map the registered buffers in the local virtual address space (lines 9-10). Similarly, this is what happens with the setting of the `vaddr` attribute of event objects `e1` and `e2`, used to synchronize the accesses to the FIFO buffer from the two sides (GN and HN). Variables `X` and `Y` then used to store the scalar values passed to specify the size of the input image. At this point, all the parameters for the actual kernel execution are ready and the `kernel_function()` can be invoked.

Listing 5: GIF FIFO sample: HN-side code

```
1  /* main.c */
```

```
 2
 3 #include "dev/mango_hn.h"
 4 #include <stdlib.h>
 5 extern void kernel_function(uint8_t *out, uint8_t *in, int X, int Y,
       mango_event_t e1, mango_event_t e2);
 6
 7 int main(int argc, char **argv){
 8     mango_init(argv);
 9     uint8_t * out = (uint8_t *)mango_memory_map(strtol(argv[5],NULL,16));
10     uint8_t * in  = (uint8_t *)mango_memory_map(strtol(argv[6],NULL,16));
11     int X = strtol(argv[7],NULL,16);
12     int Y = strtol(argv[8],NULL,16);
13     mango_event_t e1;
14     e1.vaddr = (uint32_t *)mango_memory_map(strtol(argv[9],NULL,16));
15     mango_event_t e2;
16     e2.vaddr = (uint32_t *)mango_memory_map(strtol(argv[10],NULL,16));
17     kernel_function(out, in, X, Y, e1, e2);
18     mango_close(42);
19 }
20
21 /* kernel_function.c */
22
23 #include "dev/mango_hn.h"
24 #pragma mango_kernel
25
26 void kernel_function(uint8_t *out, uint8_t *in, int X, int Y,
       mango_event_t e1, mango_event_t e2){
27     for(int i=0; i<4; i++) {
28         mango_wait(&e1, READ);
29         mango_wait(&e2, WRITE);
30         printf("KERNEL: mango_wait\n");
31         scale_frame(out, in, X, Y);
32         mango_write_synchronization(&e1, WRITE);
33         mango_write_synchronization(&e2, READ);
34         printf("KERNEL: mango_write_synchronization\n");
35     }
36 }
37
38 void scale_frame(uint8_t *out, uint8_t *in, int X, int Y){
39     int X2=X*2;
40     int Y2=Y*2;
41     for(int x=0; x<X2; x++)
42         for(int y=0; y<Y2; y++)
43             for(int c=0; c<3; c++) {
44                 out[y*X2*3+x*3+c]=in[y/2*X*3+x/2*3+c];
45             }
46 }
```

This function basically iterates to fetch and process the (four) frames of the input GIF image. The `mango_wait()` calls are used to wait for the input buffer to be ready for read access (line 28), and the output one to be ready for writing (line 29). Once the `scale_frame()` has completed its job, we can notify that the buffers are respectively accessible in write and read mode. This is done by calling `mango_write_synchronization()`. This unlocks the transfer of the next frame from the GN, and makes the output available to other kernels, representing other optional stages of the application execution.

Application Requirements

In the GIF FIFO sample application, described in Section 1.2.4, we briefly mentioned the *recipe* file, whose filesystem path is passed as the second argument of the `BBQContext` constructor. This file is an extended version of the recipe described in Section 1.1.1, used to specify the application requirements and, as we are going to see, include profiling information that can be exploited by the resource allocation policy.

The *recipe* is again an XML file, to install into the `<MANGO_ROOT>/bosp/etc/bbque/recipes/` directory. This means that a suitable procedure must be included in the compilation process of the application. Conventionally, the format file name is: "¡recipe-name¿.recipe". At run-time, the selection of the recipe file is performed by specifying the recipe name as the second argument of the `mango_init()` function, in case of C language API. Example: `mango_init(myapp-name, myrecipe)`, where we are assuming that the recipe file name is myrecipe.recipe. Otherwise, in the C++ case, the same are the arguments of the `BBQContext()` constructor.

The example provided in Listing 13 shows a typical *recipe* structure, for the MANGO API cases. The tag `application` is still the topmost level starting from which we can group the information regarding the application. A *static priority* value can be assigned by setting the `priority` attribute, where the value "0" is used to indicate the highest priority level.

The `platform` section then identifies the target system we are referring to. The recipe can include references to multiple targets, leaving to the BarbequeRTRM the burden of picking the actual system we are running on. Looking at the listed example, we can start from the target MANGO (`id`=*org.mango*). The attribute `hw`=*arch20* specifies the low-level configuration of the platform, e.g., the available processors and the topology.

Then, the section will include the specific set of *Application Working Modes* for the target platform. In general, the AWMs include the following attributes:

- A progressive numeric identifier (`id`)
- A descriptive name, which is used only for debugging/logging purposes (`name`)
- A performance level or score (`value`)
- The configuration time profiled (`config-time`) when the application switches to the given AWM

The `resources` subsections contain the resource assignment configurations. Such assignments are typically expressed in terms of CPU time quota (`pe` under `cpu`, expressed in percentage), amount of memory (`mem`), number of accelerator cores (`pe` under `acc`) and network bandwidth (`net`), in Kbps. In the example, we see a single AWM in which the resource request includes a 2-core accelerator (a processor deployed on a HN) and 1 MB of memory.

However, since in MANGO we need to have *per-task* and *per-buffer* resource mapping, the AWM description does not match completely our resource allocation requirements. The section `tg` has been introduced to this purpose: to specify the application requirements at the level of single task (kernel) (section `<reqs>`), and (optionally) to include task-graph mapping related information.

Concerning the requirements, below we list the set of possible metrics that can be currently used as attributes:

- `ctime_ms`: task (kernel) completion time in milliseconds
- `throughput_cps`: number of executions per second
- `hw_prefs`: ordered list of processing unit preferences
- `inbw_kbps`: read bandwidth
- `outbw_kbps`: write bandwidth

Listing 6: Example of application Recipe.

```xml
<?xml version="1.0"?>
<BarbequeRTRM recipe_version="0.8">
    <application priority="4">
        <platform id="org.linux.cgroup">
            <awms>
                <awm id="0" name="OK" value="100">
                    <resources>
                        <cpu>
                            <pe qty="100"/>
                        </cpu>
                        <mem qty="20" units="M"/>
                    </resources>
                </awm>
            </awms>
            <tg>
                <reqs>
                    <task name="t0" id="0"  hw_prefs="peak,gn,nup"
                        throughput_cps="2" inbw_kbps="2000" outbw_kbps="
                        2500"/>
                    <task name="t1" id="1"  hw_prefs="peak,gn,nup"
                        ctime_ms="500"/>
                </reqs>
            </tg>
        </platform>
        <platform id="org.mango" hw="arch20">
            <awms>
                <awm id="0" name="test" value="10">
                    <resources>
                        <acc>
                            <pe qty="200"/>
                        </acc>
                        <mem qty="1" units="Mb"/>
                    </resources>
                </awm>
            </awms>
            <tg>
                <reqs>
                    <task name="t0" id="0" hw_prefs="peak,gn,nup"
                        throughput_cps="1" inbw_kbps="2000" outbw_kbps="
                        1500"/>
                    <task name="t1" id="1" hw_prefs="peak,gn,nup" ctime_ms
                        ="1000"/>
                </reqs>
```

```
                    <mappings>
                        <mapping id="0" exec_time_ms="1000" power_w="20"
                            mem_bw="">
                            <tasks>
                                <task id="0" acc="0" freq_khz="0"/>
                                <task id="1" acc="1" freq_khz="0"/>
                            </tasks>
                            <buffers>
                                <buffer id="0" mem="0"/>
                            </buffers>
                        </mapping>
                    </mappings>
                </tg>
            </platform>
    </application>
</BarbequeRTRM>
```

Therefore, for each task we will have a `task` tag, including the `id` attribute, which must be consistent with the value provided at runtime in the `register_kernel()` function call; the `name` (optional) and the set of performance requirement attributes, as described above. The attributes `ctime_ms` and `throughput_cps` for instance, must be considered mutually exclusive. The former allows us to express the execution time required for a task (deadline), the latter instead focuses on the number of processing cycles completed in a second. This means that we can rely on the second when we are more interested in the rate of completed works carried out, than the actual completion time of each execution. Through the `hw_prefs` attribute, then, the developer explicitly declares the architecture supported by the task code. This means that the resource allocation policy, during the task mapping process, will look for processing units matching one of the reported architectures. The list of string values has to be considered sorted by preference, which may represent a hint for the policy. Finally, `inbw_kbps` and `outbw_kbps` are useful to allows the runtime resource allocation policy to ask for HN resource partitions, for which we can guarantee a certain data transfer bandwidth among tasks.

The application execution model, and the information exported by the platform library (HN Library), allows us to monitor the current performance comparing the runtime values with the requirements specified in the recipe. Thus, given a resource mapping choice, we can evaluate its goodness. Resource allocation policies based on static algorithms can explore the resource mapping choices space and save the results of this exploration in the `<mappings>` section of the recipe. In this section, we assume to have the set of feasible mapping choices selected by the algorithm. At run-time, the resource manager can pick one from this set, and apply it according to the current status of the system and the workload. Each mapping choice is reported under `<mapping>` and it is characterized by the profiled values of application execution time (`exec_time_ms`), peak power consumption (`power_mw`), and requirements in terms of memory access bandwidth (`mem_bw`). A mapping includes the tasks and the buffers mapping to the physical hardware resources. The attribute (`freq_khz`) allows us to specify the operating point of the target processor, wherever the DVFS support is available.

Emulation Layer

To provide developers with tools for a faster development cycle, as well as to allow demonstration and testing of the MANGOLIBS functionalities when a MANGO heterogeneous cluster is not available, MANGOLIBS includes an emulation library.

The LIBHN API provides a set of functions to allow the Barbeque Runtime Resource Manager to request, reserve and release any type of resources within the heterogeneous cluster. The MANGOLIBS emulation library (LIBGN) implements the LIBHN API and allows to perform resource allocation in a simulation mode. The three major resources – that is compute units (processors/accelerators), memory buffers located in DDR memories, and bandwidth – are under control of the local resource manager of the emulator. All these resources are stored as the internal configuration of the heterogeneous cluster and offered for reservation based on their availability and platform restrictions. To request a set of units the types of accelerators required by the application need to be indicated. To limit bandwidth utilization, the local resource manager allocates units close to the memory that will be accessed. To ensure the smallest average distance from each unit to the memory, units are selected in the nearest von Neumann neighborhood. The following three types of bandwidth resource can be reserved: bandwidth at the cluster main entry point (read and write), network bandwidth for tiles interconnect inside the FPGA cluster and memory controllers bandwidth (read and write). To avoid network and memory bandwidth usage from partitions of concurrent applications, reserved units are isolated by setting routing bandwidth on the boundary of partition to 0.

1.3 RECIPE Application Requirements

In the RECIPE project, we do not expect to introduce disruptive changes in the previously described programming models. Reasonably, what we are considering is an extension of the application requirements input, in terms of reliability and timing guarantees.

Concerning the reliability management goal, as stated among the project objectives, we aim at proposing an approach which is the most transparent possible with respect to the application developer. This means to leave the burden of managing the application checkpoint and restore to the resource manager daemon. However, some developers may still be interested in the possibility of controlling the checkpoint of its execution status at runtime. This is described in deliverable D2.2, since tightly coupled with the reliability management support, developed as extension of the local resource manager. As additional feature, we can introduce a new tag in the recipe file, through which the developer can specify a default application-specific checkpoint rate value.

For the timing requirements instead, we already shown how, especially with the MANGO programming model, the developer can set the a completion time or a throughput goal for each task (kernel) launched by the host-side code of the application. In deliverables D2.2 and D3.2, we described how we aim at meeting the timing requirements, from a probabilistic perspective. This means for example to guarantee timing constraints on the WCETs, with a probability value of p ($pWCET$). In D2.2, we explained how the pWCET is a statistical distribution. Accordingly, the recipe file can be extended with a dedicated section, devoted to providing the pWCET distribution parameters as input (offline profiling outcome). At this point, the probability value p can represent a requirement: the need of guarantee that the worst-case execution time (WCET) would remain under the expected value with a given probability value p. Or, in other terms, the possibility of tolerating WCET violations with a probability value $1 - p$.

These extension to the programming models requirements interface will be evaluated and implemented in the second half of the project.

2 Domain Specific Languages for RECIPE

To support the exploration of *Domain Specific Languages* (DSLs) in the RECIPE project, it is useful to provide a higher level interface with respect to the MANGOLIBS C/C++ API.

To this end, in RECIPE we provide Python and OpenCL bindings for MANGOLIBS. These bindings will be used to quickly explore possible extensions to the API, as well as higher level constructs that could be used in specialised variants to support an application domains needs.

As an example, the Python bindings will allow to easily manipulate an application's recipe from within the application itself, which may be useful to allow a more flexible definition of operating points to cope with unpredictable operating conditions.

The choice of Python and OpenCL is also justified by the fact that many applications in emerging domains (for HPC), such as machine learning, are developed first in Python. In particular, this is the case for RECIPE UC3. By providing Python bindings, we make the RECIPE stack more attractive for developers who are entering now the HPC domain, and may not be familiar with C++ or Fortran.

2.1 Python MANGOLIBS API

The baseline Python API is defined as follows. It is currently implemented as a mock-up that allow functionally correct execution of application, but does not actually interact with the resource manager or the LIBHN.

Context The `BBQContext` class mirrors the corresponding class of the C++ API. It allows the definition of a context for the current host-side application, including passing a resource management recipe.

```
class BBQContext(object):
    """ A class to hold the current state of the host-side runtime."""
    def __init__(self, name="app", recipe="generic"):
        """Pass the desired recipe to the RTRM"""

    def resource_allocation(self,tg):
        """Resource Allocation for a task graph of the application
        @param tg The task graph to allocate resources for
        @returns An exit code signalling the correct allocation (or not)
        @note Current implementation is a dummy."""

     def resource_deallocation(self):
        """Resource Dellocation for a task graph of the application
        @returns An exit code signalling the correct allocation
    (or not)
        @note Current implementation is a dummy."""

    def start_kernel(self, kernel, args, ev=None):
        """Start a given kernel
        @param kernel The Kernel to start
        @param args A KernelArguments object which provides the
    marshalling of
        arguments
```

Listing 7: Example of application Recipe.

Kernels The `Kernel` class mirrors the equivalent class in the C++ API. It allows the definition of a kernel characterised by a specific function, as well as a set of input and output buffers.

```
class Kernel(object):
    """@brief Kernel descriptor"""
    def __init__(self,kernelID, kfunction, buff_in=[], buff_out
  =[]):

    def is_a_reader(self,buff_id):

    def is_a_writer(self,buff_id):

    def get_termination_event(self):

    def get_task_events(self):

    def get_virtual_address(self):

    def get_physical_address(self):

    def get_mem_tile(self):

    def get_kernel(self):

    def get_id(self):

    def set_thread_count(self,tcount):

    def get_thread_count(self):
```

Listing 8: Example of application Recipe.

Buffers The `Buffer` class mirrors the equivalent class in the C++ API. It allows the definition of a memory buffer on the HN node memory characterised by its size in bytes, as well as a set of readers and writers (kernels).

```
class Buffer(object):
  """HN shared memory buffer descriptor """
  def __init__(self,buffID, size, kern_in=[], kern_out=[]):

  def write(self,GN_buffer, size, global_size=0):
    """Memory transfer from GN to HN in DIRECT mode
 @param GN_buffer A memory buffer in the GN address space
 This function performs a copy between a memory region in the GN address space
    and one in the HN address space. The copy works on the entire buffer size
 specified in the HN buffer descriptor.
 @note Current specification assumes synchronous transfer
    @note Current implementation uses a specialised GNBuffer class"""

  def read(self, GN_buffer, size, global_size=0):
    """Memory transfer from HN to GN in DIRECT mode
 @param GN_buffer A memory buffer in the GN address space
 This function performs a copy between a memory region in the HN address space
 and one in the GN address space. The copy works on the entire buffer size
 specified in the HN buffer descriptor.
 @note Current specification assumes synchronous transfer
    @note Current implementation uses a specialised GNBuffer class"""

  def resize(self,size):

  def get_id(self):

  def get_size(self):

  def get_phy_addr(self):

  def get_mem_tile(self):

  def set_phy_addr(self,phy_address):

  def get_event(self):

  def __len__(self):
```

Listing 9: Example of application Recipe.

Events The `Event` class mirrors the equivalent class in the C++ API. It allows the definition of events, associated to memory locations in the HN node mutual exclusion registers (which are memory mapped in the HN physical address space).

```python
class Event(object):
    """HN Event descriptor
    These events are also used at the GN level."""
    def __init__(self, _event, kernel_id_in=[], kernel_id_out
  =[]):

    def wait(self):

    def signal(self):
```

Listing 10: Example of application Recipe.

Task Graph The `TaskGraph` class provides a simplified interface with respect to the C++ API, since the lists of kernels, buffers and events can be updated directly using the standard Python `list` operators.

```python
class TaskGraph():
    """A structure type representing a task graph
    Note The actual graph structure can be inferred by reading
  the
    mango.Buffer and mango.Event objects, which include links
  to the readers
    and writers."""
    def __init__(self, kernels=[], buffers=[], events=[]):
        """Define a task graph
        @param kernels list of mango.Kernel objects
        representing the kernels in the task graph
        @param buffers list of mango.Buffer objects
        representing the buffers in the task graph
        @param events list of mango.Event objects
        representing the synchronization events in the task
  graph
```

Listing 11: Example of application Recipe.

Kernel Functions The `KernelFunction` class mirrors the behaviour of the corresponding C++ class. It allows to define different implementations of a kernel function to run on different types of compute units. In the current mock-up implementation, only kernels running on the emulator node are allowed, and must be Python code fragments returning the kernel function.

```python
class KernelFunction(object):
    """Kernel function
    This is an array of function pointers to support multiple versions of the
    kernel.
    Note: This allows one kernel implementation per type of unit. If more are
    desired, we need to redesign this data structure."""
    def __init__(self):

    def load(self, kernel,unittype):

    def get_kernel_version(self, unittype):

    def set_kernel_size(self,unittype, size):

    def get_kernel_size(self,unittype):

    def is_loaded(self):

    def __len__(self):
```

Listing 12: Example of application Recipe.

Example Application

The following (very simple) application example can be functionally run with the mock-up implementation of the Python API. It performs a pair-wise addition between the elements of two buffers (lists of integers).

An additional `mango.GNBuffer` class is used to wrap the lists on the host side.

```python
import mango

kernel="""def mango_kernel():
    def kernel(v1, v2, v3):
        for i,j in zip(v1,v2):
            v3.append(i*j)
        return
    return kernel

kernel=mango_kernel()
"""

cxt=mango.BBQContext()
kf=mango.KernelFunction()
```

```
kf.load(kernel,"gn")
k1=mango.Kernel(1,kf)
b1=mango.Buffer(1,size=10)
b2=mango.Buffer(2,size=10)
b3=mango.Buffer(3,size=10)
arg1=mango.Arg(1,b1,len(b1))
arg2=mango.Arg(2,b2,len(b2))
arg3=mango.Arg(3,b3,len(b3))
tg=mango.TaskGraph([k1],[b1,b2,b3])
cxt.resource_allocation(tg)
b1.write(mango.GNBuffer(range(1,10)),10)
b2.write(mango.GNBuffer(range(1,10)),10)
ev=cxt.start_kernel(k1,[arg1,arg2,arg3])
ev.wait()
out=mango.GNBuffer()
b3.read(out,10)
print(out.data)
```

Listing 13: Example of application Recipe.

2.2 OpenCL MANGOLIBS API

In order to properly support widespread programming languages choices, in RECIPE we aim also at providing a suitable OpenCL wrapper for the MANGOLIBS host-side API.

The MANGOLIBS OpenCL API in particular, would represent and extension of the OpenCL standard, due to the need of supporting the concept of Task Graph, as well as mapping the main constructs of OpenCL on the *libmango* C++ API. The following constructs and features from OpenCL are supported in our current implementation:

Platform : The Platform model of OpenCL is composed of a host and one or more OpenCL devices. Currently, and embedded profile is used, as no online compilation is currently provided by the MANGO API implementation.

Device : MANGO employs internally a similar view of the architecture as that exposed by OpenCL. However, this view is opaque, since the devices are managed by the resource manager. As a result, device selection is not up to the programmer, and is delayed to the resource allocation phase. Consequently, we offer a managed device, which is the only device available in the MANGO platform.

Context : The OpenCL Context maps directly to the MANGO Context.

Buffer Object : Buffer Objects are mapped to MANGO Buffers. FIFO Buffers are not used by the OpenCL Wrapper.

Program Object : the Program Object in OpenCL maps directly on the MANGO Kernel-Function, except that multiple implementations can be provided in MANGO.

Kernel Object : corresponds to MANGO Kernel. As the method of setting arguments differs, the OpenCL Wrapper keeps track of the kernels arguments to update the MANGO kernel argument every time an OpenCL call for setting arguments is performed.

Events : OpenCL events map directly onto MANGO events.

CommandQueue : the queue contains the kernels to be executed, ordered by submission time. This corresponds to the TaskGraph in MANGO, which however has an explicit way to detect dependencies. As a result, the TaskGraph is not wrapped, and must be explicitly managed even in OpenCL.

Exceptions : Generally, each MANGO API Error code can be mapped to an OpenCL exception.

A typical OpenCL program works through the following flow:

- Get available Platform: `clGetPlatformIDs()`
- Get available Devices: `clGetDeviceIDs()`
- Create Context: `clCreateContext()`
- Create Command Queue: `clCreateCommandQueue()`
- Create Buffers: `clCreateBuffer()`
- Create and Build Program: `clCreateProgramWithSource()` or `clCreateProgramWithBinary()`[3]
- Create Kernel: `clCreateKernel()`
- Set Kernel Arguments: `clSetKernelArg()`
- Queue Buffers: `clEnqueueWriteBuffer()`
- Queue and execute Kernels: `clEnqueueNDRangeKernel()` or `clEnqueueTask()`
- Read the result from read buffer: `clEnqueueReadBuffer()`
- Release all resources, program, kernel, buffers and context: `clReleaseProgram()`; `clReleaseKernel()`; `clReleaseMemObject()`

The workflow is implemented in MANGO as follows:

[3]`mangolibs` currently implements only the latter.

OpenCL function	MANGO Implementation
clGetPlatformIDs()	Returns the MANGO platform
clGetDeviceIDs()	Returns a special managed device
clCreateContext()	Maps to mango_init()
clCreateCommandQueue()	Creates an empty queue, which will be populated by mango_task_graph_create()
clCreateBuffer()	mango_register_memory()
clCreateProgramWithBinary()	mango_kernelfunction_init(), mango_load_kernel()
clCreateProgramWithSource()	mango_kernelfunction_init() mango_load_kernel()[4]
clCreateKernel()	mango_register_kernel()
clSetKernelArg()	mango_set_args()
clEnqueueWriteBuffer()	mango_write()
clEnqueueTask()	mango_start_kernel()
clEnqueueReadBuffer()	mango_read()
clReleaseKernel()	mango_deregister_kernel()
clReleaseMemObject()	mango_deregister_memory()

Table 1: OpenCL to MANGO API function calls mapping.

Example Application

In the following Listing 14, we show an example of code (Matrix Multiplication) implemented in OpenCL and using the MANGO API, for the specific requirements of resource allocation. The OpenCL API wraps the underlying MANGO API according to Table 1.

```
// Example: Matrix Multiplication (Host-side code)
void main(int argc, char**argv) {
    int *A, *B, *C, *D, rows, columns, out=0, i, j;
    cl_int errNum, plid, nplat, devid, ndevs;
    cl_context context = NULL;
    cl_command_queue queue = NULL;
    unsigned char * binaries[1];
    binaries[0] = loadfromfile("./test_gn_app", &lenghts);
    rows = atoi(argv[1]);
    columns = atoi(argv[2]);
    /* matrix allocation */
    A = malloc(rows*columns*sizeof(int));
    B = malloc(rows*columns*sizeof(int));
    C = malloc(rows*rows*sizeof(int));
    D = malloc(rows*rows*sizeof(int));
    /* input matrices initialization */
    init_matrix(A, rows, columns);
    init_matrix(B, rows, columns);
    /* initialization of the mango context */
    clGetPlatformIDs(1, &plid, &nplat);
    clGetDeviceIDs(plid, NULL, 1, &devid, &ndevs);
    context = clCreateContextFromType(NULL,
    CL_DEVICE_TYPE_CPU, NULL, NULL, &errNum);
    queue=clCreateCommandQueue(context, devid, 0, NULL);
    /* kernel creation */

    cl_program *k = clCreateProgramWithBinary(context, 1, lengths,
    binaries, NULL, NULL);
    cl_kernel k1 = clCreateKernel(k, NULL, &errNum);
    /* registration of buffers */
    cl_mem b1 = clCreateBuffer(k1, CL_MEM_READ_ONLY, rows*columns*sizeof(
    int), NULL,&errNum);
    cl_mem b2 = clCreateBuffer(k1, CL_MEM_READ_ONLY, rows*columns*sizeof(
    int), NULL,&errNum);
    cl_mem b3 = clCreateBuffer(k1, CL_MEM_WRITE_ONLY, rows*rows*sizeof(int
    ), NULL,&errNum);
    /* Registration of task graph */
    mango_task_graph_t *tg = mango_task_graph_create(1, 3, 0, k1, b1, b2,
    b3);
    /* resource allocation */
    mango_resource_allocation(tg);
    /* Execution preparation */
    clSetKernelArg(*k1, 0, sizeof(uint64_t), &rows);
    clSetKernelArg(*k1, 1, sizeof(uint64_t), &columns);
    clSetKernelArg(*k1, 2, sizeof(cl_mem), &b1);
    clSetKernelArg(*k1, 3, sizeof(cl_mem), &b2);
    clSetKernelArg(*k1, 4, sizeof(cl_mem), &b3);*/
    /* Data transfer host->device */
    clEnqueueWriteBuffer(queue, b1, 0, 0, 0, A, 0, NULL, NULL);
    clEnqueueWriteBuffer(queue, b2, 0, 0, 0, B, 0, NULL, NULL);
    cl_event ev;
    /* spawn kernel */
    errNum = clEnqueueTask(queue, k1, 1, NULL, NULL, NULL, 0, NULL, &ev);
    /* reading results */
    clEnqueueReadBuffer(queue, b3, 0, 0, 0, C, 1, ev, NULL);
    /* shut down the mango infrastructure */
    mango_resource_deallocation(tg);
    mango_task_graph_destroy_all(tg);
    clReleaseContext(context);
}
```

Listing 14: Example of Matrix Multiplication using OpenCL on top of MANGO

2.3 Dynamic Compilation of Kernels

In MANGO, the MANGOLIBS API are defined as able to accept source kernels. However, this capability was not implemented due to resource constraints, as well as and limited applicability in the context of the MANGO use case scenarios. In RECIPE, dynamic kernel compilation can be useful to apply different operating points through compiler transformations (e.g., loop unrolling). This requires the ability to access the recipe for the specific application from MANGOLIBS (currently, it is only used by the runtime manager, or to receive the same information from the runtime manager itself).

This capability will be implemented in the MANGOLIBS during the first quarter of year 3. The dynamic compilation method will not modify the programming interface, but will require some extensions to the current representation of the platform information. A configuration file contains the mapping between the architecture types available in the MANGOLIBS and the paths to their compilers (which include the necessary source-to-source transformations needed to perform the argument decoding). This configuration file is loaded into a map data structure at the initialization of the MANGOLIBS, either through `mango_init` or through the creation of the `mango::Context`. The kernel loading function will then access this information to perform the selection of the appropriate compiler for the selected accelerator platform.

The key advantage of dynamic compilation is the ability to perform a re-compilation when different parameters are required – for example because the kernel needs to be re-allocated on a variant of the same platform with different capabilities. In this case, the code can be recompiled using a different optimization strategy (possibly including specialisation based on runtime constants) to achieve better performances.

3 Heterogeneous Acceleration Programming

This section deals with the low-level OpenCL programming of heterogeneous accelerators, with emphasis on the specific features of the target architectures adopted in RECIPE. We recall that Deliverable 4.1 set as the outcome of Task 4.1 the spectrum of approaches to the integration of heterogeneous acceleration in the RECIPE platform, depending on the specific use case requirements:

- full custom HDL implementation, suitable for performance-critical, relatively simple and regular kernels;
- optimized library-based design, for well-supported kernels to be implemented in hardware, e.g. linear algebra;
- pure-hardware HLS-based design, for non-standard kernels or control-intensive parts of the application that are not performance-critical;
- software-programmed accelerators, particularly the nu+ vector core and associated LLVM-based compiler imported from MANGO, which is suitable for control-intensive parts of the software application that do not match the restrictions of HLS and/or data-intensive kernels benefitting from vector-style programming.

These approaches essentially correspond to different types of heterogeneous acceleration resources that can be found in the platform, particulary in heterogeneous fabrics based on Field-Programmable Gate Arrays. Of the above four types of acceleration resources, the first two represent components that need low-level physical integration in the system. For that, we are developing in WP4 a customizable reference design for easy configuration of an FPGA system integrating full-custom or library IPs. The remaining two types, software-programmable and HLS accelerators, on the other hand, lend themselves to a higher-level abstraction and a corresponding programming interface, that can be profitably unified under a common programming model specialized for accelerator programming. In RECIPE, we identified the OpenCL model as a good fit for this purpose. In Task 2.3, we thus allocated a significant effort to:

- implementing the OpenCL support for the configurable GPU-like soft-core, called *nu+*, developed by the MANGO H2020-FETHPC project;
- carefully evaluating programming styles and specificities of OpenCL as supported by HLS tool vendors.

This section provides the detail of the work done for the implementing the *nu+* OpenCL support, to be used as a general-purpose programming approach for the integration of customizable accelerators, in line with the MANGO philosophy. Furthermore, because proprietary FPGA-targeted OpenCL tools require particular programming styles that deviate from the common use of OpenCL in GPU-like accelerators, the section also briefly describes the main guidelines to the optimization of OpenCL-based FPGA kernels, particularly referring to the case of Xilinx tools which are of interest for the RECIPE prototype.

3.1 Implementation of OpenCL for the nu+ GPU-like programmable core

The nu+ GPU-like configurable core imported from MANGO exposes useful features for improved resource efficiency in that it provides an abundance of threads executing in a SIMD-like fashion, while reducing control overheads and hiding possibly long operation latencies. Accelerators based on nu+ can effectively exploit multithreading, SIMD/SIMT operation, and low-overhead control flow constructs, in addition to a range of advanced architecture customization capabilities, in order to enable a very high-level utilization of the underlying resources. RECIPE imported from MANGO the baseline implementation of nu+, ensuring a set of minimum features including: support for hardware multithreading; data-level parallelism through large-size vector/SIMD/SIMT support; multiprocessor organization allowing non-SIMT execution; lightweight control flow constructs exposed to the programmer, such as predication and mechanisms for optimizing diverging threads and improving datapath utilization; hybrid memory hierarchy providing both coherent caches and noncoherent scratch-pad memory; on-tile performance counters, e.g. for utilization, stalls, instruction counters; facilities for fault/interrupt handling and debug. In RECIPE, particularly in Task 2.3, an OpenCL (version 1.1) implementation was built on top of this hardware system.

OpenCL model on nu+

The OpenCL framework is defined in terms of a set of models describing the way the target features meet its specification. Bearing in mind the custom design of nu+, it is essential to explain how physical resources are mapped to the OpenCL models. The OpenCL framework is defined in terms of a set of models describing the way the target features meet its specification. Bearing in mind the custom design of nu+, it is essential to explain how physical resources are mapped to the OpenCL models.

OpenCL Platform Model on nu+. From the OpenCL specification, a platform is defined as a set of compute devices on which the host-system is connected to. Each device is further divided into several compute units (CUs), each of them defined as a collection of processing elements (PEs).
Referring to nu+ core, it is structured in terms of a set of eight hardware threads. Each hardware thread races with each other to access the sixteen hardware lanes.

OpenCL Specification	Hardware Feature
Compute Device (CD)	nu+ System
Compute Unit (CU)	nu+ Core
Processing Element (PE)	nu+ Hardware Thread

Table 2: Platform Model Mapping

Figure 3 and Figure 2 show how the OpenCL Platform model is adapted to the target. As it is highlighted, the compute device abstraction is physically mapped on the *nu+ System*. Each *nu+ Core* maps on the OpenCL Compute Unit. Internally, the core is structured into *hardware threads*, each of them representing the abstraction of the OpenCL processing element.

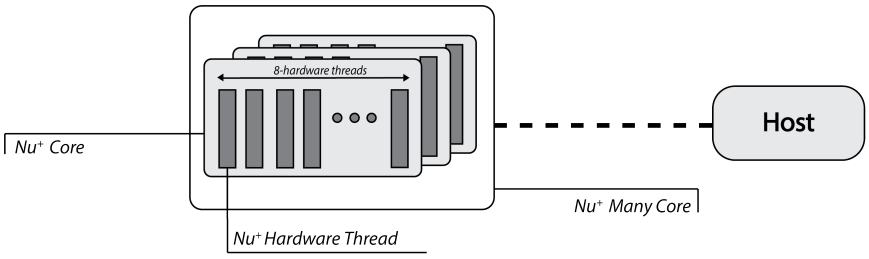

Figure 3: Platform Model Mapping - Visual

OpenCL Execution Model on nu+. From the execution model point of view, OpenCL relies on an N-dimensional index space, where each point represent a kernel instance execution. Since the physical kernel instance execution is done by the hardware threads, the OpenCL work-item is mapped on a nu+ single hardware-thread. Consequently, a work-group is defined as a set of hardware threads, and all work-items in a work-group execute on a single compute unit, that is the nu+ Core as depicted in Figure 3. Figure 4 and Figure 3 illustrate the mapping explained above.

OpenCL Specification	Hardware Feature
Kernel Instance	nu+ System
Work-Group	nu+ Core
Work-Item	nu+ Hardware Thread

Table 3: Execution Model Mapping

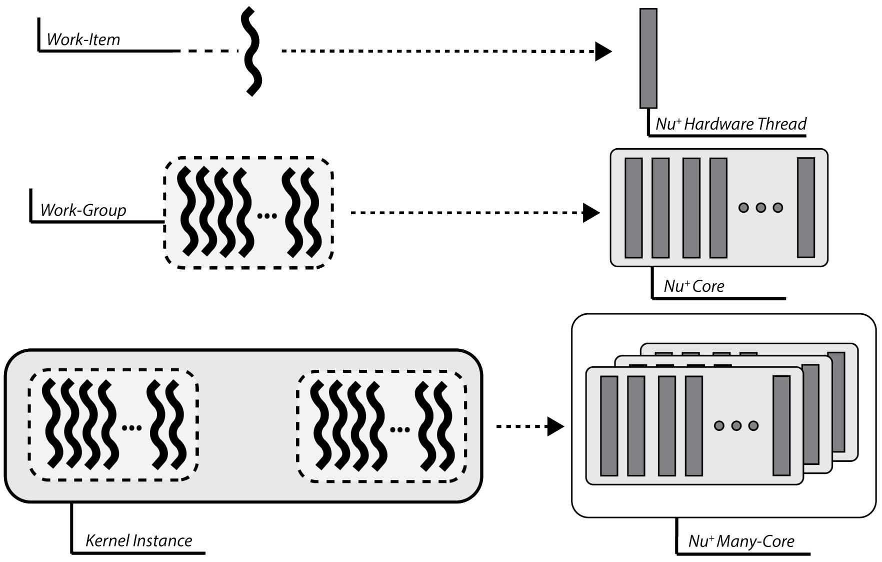

Figure 4: Execution Model Mapping - Visual

OpenCL Memory Model on nu+. OpenCL provides logical sectioning of the device memory by defining constraints on variable scope regarding which execution model elements have access to.
OpenCL distinguishes among four kinds of address spaces: the *global* and *constant* ones, that

can be accessed by all work-items in all work-groups, *local* one, that is only visible to work-items within a work-group and the *private* one, which can only be accessed by the single work-item.
The target-platform is provided with a DDR memory, that is the *device memory* in OpenCL nomenclature. Consequently, variables are physically mapped to this memory. The compiler itself verifies if the OpenCL constraints are satisfied by looking at the address-space qualifier.

The nu+ architecture provides a per-core *ScratchPad Memory* that is a noncoherent memory area which can be exclusively accessed by each core. While this memory is compliant with the OpenCL *local* memory features, in the current platform implementation the notions of *local* and *global* memory are merged into the *global* memory.
Furthermore, each hardware thread within the nu+ core may rely on a private stack area. This memory section is private to each hardware thread, that is the OpenCL work-item, and cannot be addressed by others. As a result, each stack area acts as the OpenCL *private memory.* A summary is provided in Figure 5 and Figure 4.

OpenCL Specification	**Hardware Feature**
Private Memory	nu+ Stack Area
Local Memory	nu+ DDR Memory
Global Memory	nu+ DDR Memory

Table 4: Memory Model Mapping

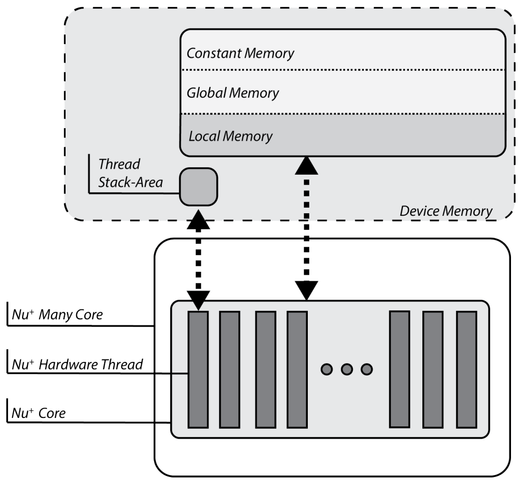

Figure 5: Memory Model Mapping

OpenCL Programming Model on nu+. OpenCL supports two kinds of programming models, *data-* and *task-parallel.*
A data-parallel model requires that each point of the OpenCL index space is executing a kernel instance. Since each point represent a work-item and work-items are mapped to the hardware threads, the data-parallel requirements are correctly satisfied.
A task-parallel programming model requires that each kernel instance is independently executed in any point of the index space. In this case, each work-item is not constrained to executing the same kernel instance as others. Since each nu+ hardware thread can rely on a set of sixteen

hardware lanes, OpenCL support is realized with the aim to use vector types.
The following list shows how lock-step execution is supported.

- `char`n, `uchar`n are respectively mapped to `vec16i8` and textttvec16u8, where n=16 (while only n = 16 is supported, other datapath configurations could support different number of lanes).
- `short`n, `ushort`n are respectively mapped to `vec16i16` and textttvec16i32, where n=16.
- `int`n, `uint`n are respectively mapped to `vec16i32` and textttvec16u32, where n=16.
- `float`n are mapped to `vec16f32`, where n=16.

Designing OpenCL Host-Runtime on nu+

The OpenCL Runtime implements the set of functions used to coordinate and handle the device, to start applications and to control their execution. The design of OpenCL Runtime on nu+ relies on a two-level layered architecture, implementing the following hierarchical abstraction:

- A *low-level* abstraction, made up of the OpenCL UML Class Diagram implementation;
- A *high-level* abstraction, formed of APIs implemented on lower-level methods.

Developers only access the API level, without having to deal with low-level mechanisms for direct device interfacing.

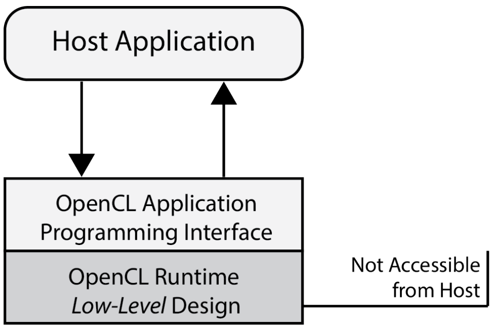
Host Application
OpenCL Application Programming Interface
OpenCL Runtime *Low-Level* Design
Not Accessible from Host

Figure 6: Runtime Layered Design

Low-Level Runtime Design. Low-level runtime is designed in an object-oriented fashion consistently with the OpenCL API specification providing a simple map between the APIs and the underlying object representation.

Figure 7 shows the implemented structure as a class diagram. *nu+* hardware features are not capable to fulfill the entire specification. Consequently, the *Image* class is not considered in the implementation, as well as the *Sampler* class.
All classes inherit from `CLObject`, that is an abstract class that holds common information. The interface to devices is implemented in a set of functions that are collected in the `DeviceInterface` header file in order to provide a centralised way to access and to manage the devices.

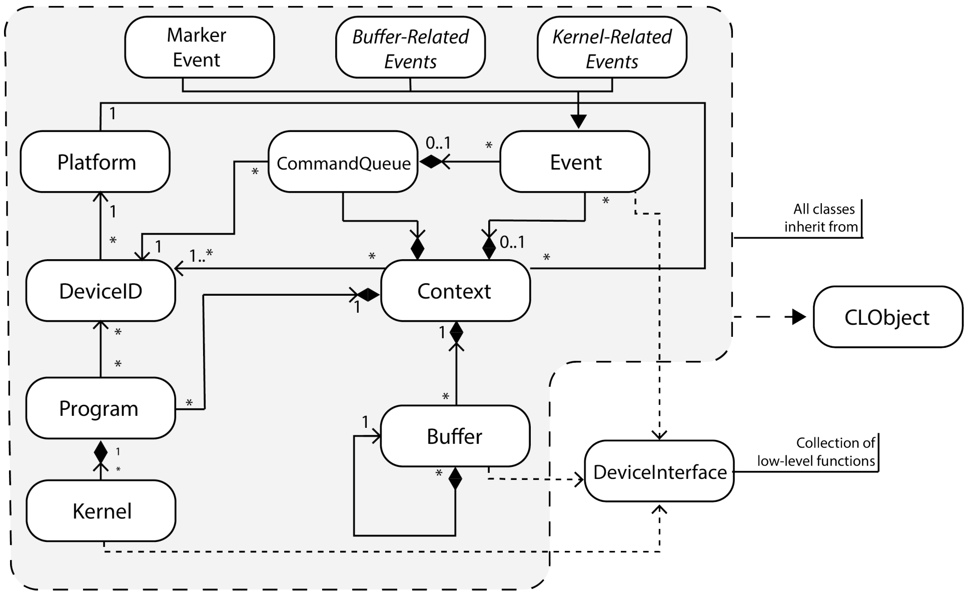

Figure 7: OpenCL low-level runtime design

High-Level Application Programming Interface Design. The API implementation is based on `C++11` and is provided in the form of a dynamic linkable library to be directly linked into the final executable.

Figure 8 shows the interaction between OpenCL APIs and the related low-level runtime. The communication paradigm adopted is of *request-response* type. Consequently, when the host program calls an OpenCL function, the underlying object representation handles the request. The API function waits for the response and sends it to the host, adapting the format to be compliant with the OpenCL specification.

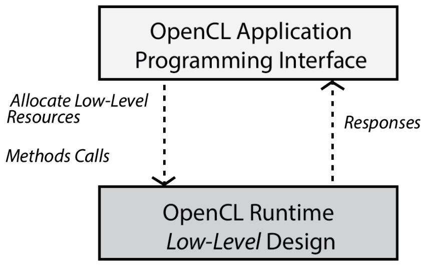
OpenCL Application Programming Interface
Allocate Low-Level Resources
Methods Calls
Responses
OpenCL Runtime Low-Level Design

Figure 8: Interaction between *low-level* runtime and *high-level* APIs

Compilation Support to OpenCL C Kernels

The execution of OpenCL C kernels on the custom device requires modifications related to the compilation flow, regarding both the front-end and the back-end phases.
Clang, the LLVM frontend, natively supports the compilation of OpenCL C kernels. However, to build an executable OpenCL C memory image for the target platform it is necessary to define the support to vector types and the implementation of OpenCL C builtins.

Vector Types. OpenCL provides a set of vector types to explicitly support the lock-step execution. Bearing in mind the nu+ structure, only the following types may be supported:

- `char16`, `uchar16` are respectively mapped to `vec16i8` and `vec16u8`;
- `short16`, `ushort16` are respectively mapped to `vec16i16` and `vec16i32`
- `int16`, `uint16` are respectively mapped to `vec16i32` and `vec16u32`;
- `float16` is mapped to `vec16f32`.

OpenCL support for these vector types is provided by adapting the compiler standard library. The adaption involves the modification of the `stdint.h` library, by adding the lines showed in Figure 9. However, to ensure better compatibility between native and OpenCL C kernels, a dedicated `stdint` library has been defined.

```
typedef char char16 __attribute__((ext_vector_type(16)));
typedef unsigned char uchar16 __attribute__((ext_vector_type(16)));
typedef short short16 __attribute__((ext_vector_type(16)));
typedef unsigned short ushort16 __attribute__((ext_vector_type(16)));
typedef int int16 __attribute__((ext_vector_type(16)));
typedef unsigned int uint16 __attribute__((ext_vector_type(16)));
typedef float float16 __attribute__((ext_vector_type(16)));
```

Figure 9: `stdint.h` library adaption

Work-item Builtins. OpenCL supports a set of builtins used by work-items to get information of the execution, such as work-item local and global IDs, index space sizes and so on, as shown in Figure 10.

```
uint get_global_id(uint dimindx){
 return __builtin_nuplus_read_control_reg(GLOBAL_ID);
}

uint get_group_id(uint dimindx){
 return __builtin_nuplus_read_control_reg(CORE_ID);
}
```

Figure 10: Chunk of `stdlib.c` containing *work-item builtins* implementation

LLVM IR Transformation. In order to be capable of OpenCL C kernel execution, the nu+ LLVM back-end needs to be modified to allow the generation of the necessary adaption code. The target *Start Routine* is designed to have a single entry-point, the *main* function, that is in charge of parsing parameters and to explicitly call the kernel function. The entry-point of an OpenCL C kernel is the function itself. The adopted solution to fill this semantic gap relies on the definition of a LLVM IR Module Pass.

The generated code should look like the one showed in Figure 11.

As a consequence, the `ModulePass` should be structured in the following parts:

```
; Function Attrs: nounwind
define void @main(i32, i8** nocapture readonly) local_unnamed_addr #0 {
  %3 = bitcast i8** %1 to i32*
  %4 = load i32, i32* %3, align 4
  ...
  tail call void @kernel_f(i32 %4, ...) #2
  ret void
}
```

Figure 11: Example of the Module Pass desired output

- Definition of the `main` function signature:

  ```
  FunctionType *main_type =
  TypeBuilder<void(int, char**), false>::get(ctx);
  Function *func =
  cast<Function>(M.getOrInsertFunction("main", main_type));
  ```

- Creation of an `entry` labeled basic block:

  ```
  BasicBlock *block = BasicBlock::Create(ctx, "entry", func, 0);
  builder.SetInsertPoint(block);
  ```

- For each kernel argument, create an aligned load instruction to retrieve the argument from the `argv` array and add it to the kernel arguments:

  ```
  argument_list[counter] = builder.CreateLoad(arg.getType(),
  builder.CreateBitCast(builder.CreateGEP(&argv,
  builder.getInt32(index)),
  arg.getType()->getPointerTo()));
  argument_list[counter]->setAlignment(4);
  kernel_arguments.push_back(argument_list[counter++]);
  ```

- Kernel function call:

  ```
  builder.CreateCall(kernel_function, kernel_arguments);
  ```

Once the pass is defined, it has to be registered in the `PassManager`, to add it to the IR optimization pipeline.

```
__kernel void kernel_function(unsigned param,
  __global unsigned* out)
{

 unsigned int i = 0;
 *out = i + param;

}
```

Figure 12: OpenCL kernel function

```
define void @main(i32, i8** nocapture readonly) local_unnamed_addr #1 {
entry:
  %2 = getelementptr i8*, i8** %1, i64 6
  %3 = bitcast i8** %2 to i32*
  %4 = load i32, i32* %3, align 4
  %5 = getelementptr i8*, i8** %1, i64 7
  %6 = bitcast i8** %5 to i32**
  %7 = load i32*, i32** %6, align 4
  tail call void @kernel_function(i32 %4, i32* %7)
  ret void
}
```

Figure 13: Generated `main` function

For instance, Figure 13 and Figure 12 show the execution of the *main* code generation. Figure 12 presents an example of a simple OpenCL C kernel. Figure 13 shows the IR-format of the generated main function.

3.2 HLS-targeted OpenCL programming interface

FPGA accelerators have been traditionally designed by means of Hardware Description Languages (HDLs), such as VHDL or Verilog. HDL-based design requires an extensive knowledge about hardware-design patterns as well as an intensive testing and simulation to check whether the generated design is correct, both in terms of functional correctness, and in the meeting of hardware constraints. Recently, solutions based on high-level languages, known as High Level Synthesis (HLS) flows, have become a consolidate approach to the implementation of hardware functions avoiding the difficulties of HDL design. Nevertheless, an FPGA-based accelerator architecture also includes the designing of communication interfaces with the host and memories. Overcoming these limitations requires the implementation of a framework that includes firmware, software and device drivers to connect, control and transfer data to and from the FPGA. For this purpose, both Xilinx and Intel FPGA support OpenCL-based design tools, presenting them as a more abstract design entry solution compared to direct use of HLS (although HLS is still required behind the scenes). While C/C++ programs used with HLS require specific pragmas to control low-level aspects, e.g. memory interface, pipelining, etc, OpenCL kernels do not strictly require additional annotations, unless the developer is seeking improved optimization. That however makes OpenCL programming less performance-friendly than direct HLS, making its use uncertain in cases where the full potential of FPGAs is to be evaluated and compared to alternative acceleration platforms. Furthermore, while in principle OpenCL is meant to provide a shared model across heterogeneous architectures, ranging from GPUs to FPGAs, the actual execution models that are exploited in the two cases remain different. This divergence essentially boils down to the difference between vector-like parallelism and pipeline parallelism which are exploited in the two cases, plus a number of features that are specifically aimed for FPGA design, e.g. heterogeneous memory support, channels, etc. In conclusion, while OpenCL for GPUs and for HLS-based FPGA design can in principle ensure functional portability across architectures, it is very likely that, for practical purposes, the code needs to be re-written when moving from one type of architecture to the other.

Based on the above observations, RECIPE considered the evaluation of OpenCL for HLS-based FPGA design, although not as a primary choice. In Task 2.4, thus, we also allocated some effort to evaluating the Xilinx OpenCL support, relying on the prototype platform built in WP4 and working with self-contained acceleration kernels exercising the potential of customized FPGA acceleration. Hence, we came up with a set of design guidelines for the HLS-targeted use of the OpenCL programming interface, which are part of of this deliverable and are complementary to the more general and portable support provided by the GPU-like programmable core, presented in the previous subsection.

OpenCL kernel design flow for FPGAs

The development of acceleration functions through OpenCL SDK for FPGAs relies on two main components: an FPGA bitstream containing the implemented acceleration function, and a host program to manage the FPGA-based accelerator.

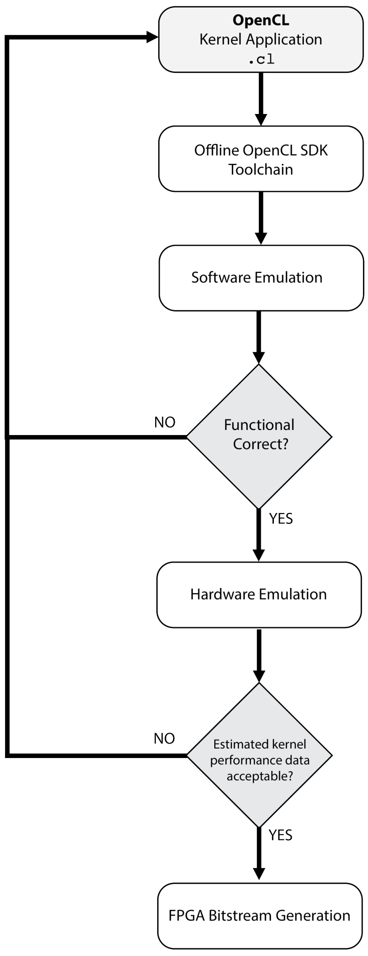
OpenCL
Kernel Application
.cl
Offline OpenCL SDK Toolchain
Software Emulation
NO
Functional Correct?
YES
Hardware Emulation
NO
Estimated kernel performance data acceptable?
YES
FPGA Bitstream Generation

Figure 14: Acceleration function design flow

Figure 14 shows a simplified view of a typical kernel design flow over the OpenCL SDK for FPGAs. Two decision points are highlighted: the first one checks the functional correctness of the algorithm, after performing a CPU-based emulation. If the functional correctness is not verified, kernel design modifications are required. Otherwise, the next step is the hardware emulation and the generation of reports containing information on estimated performance. If the estimated data satisfies the performance constraints, the FPGA bitstream generation starts. Otherwise, kernel re-design could be required.

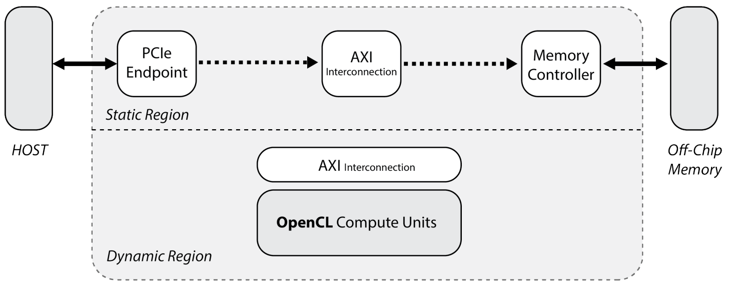

Figure 15: Internal Structure of a OpenCL-enabled device

An OpenCL-enabled FPGA device is internally structured in the following regions:

- A **Static Region**, containing the implementation of the communication logic for the off-chip DDR memory, the *Dynamic Region*, and the PCIe endpoint reserved for the host interconnection. This region can be flashed onto the an EEPROM on the board.
- A **Dynamic Region**, consisting of a reconfigurable logic area in which OpenCL-generated compute units are placed. This region relies on an AXI interconnection to interface the static-region for both memory and host accesses.

The FPGA bitstream generation phase produces the compute units to be placed in the dynamic region. Memory- and host-interfaces are automatically generated by the tool. Figure 15 shows the structure of a OpenCL-enabled device.

Improving Performance of OpenCL kernels

To leverage computational parallelism during the implementation of an algorithm on the FPGA, the use of OpenCL extension attributes is required. The following subsections provide best practices for the development of OpenCL kernels, based on Xilinx SDAccel environment, that are of potential relevance for RECIPE.

Exploiting Data Parallelism. The easiest way to exploit data parallelism is to use OpenCL vector types. The use of vector type kernel arguments allows control over the AXI interface data width. To maximize the data throughput, it is recommended to choose data types mapping to the full data width on the memory controller. This enables the optimization of burst transfers through coalesced memory accesses.

```
__kernel AIO_Stencil ( __global restrict float16 *in,
                        ...
                        ) {}
```

The above kernel definition shows the way to define a vector data type argument. This declaration tells the compiler to set a 512-bit interface between the host and the memory bank where data pointed by `in` is stored. It is important to highlight the usage of the attribute `restrict` telling the compiler that no other memory interfaces have to be generated to access data pointed by `in`.

Besides the usage of vector types, the OpenCL standard natively supports the partitioning of the execution in work-groups and work-items. This behavior has to be statically defined prior to kernel definition using the attribute `reqd_work_group_size(x,y,z)`, where `x,y,z` represent the sizes of the work-group. For example, the following code uses a single work-group composed by N work-items.

```
__attribute__((reqd_work_group_size (N,1,1))
__kernel AIO_Stencil ( __global restrict float16 *in,
                        ...
                        ) {}
```

Loop Parallelism. FPGA-targeted OpenCL flows, like the Xilinx SDAccel environment, allow improvement of loop performance by using two different approaches: loop unrolling and loop pipelining.

Loop Unrolling. Unrolling a loop enables the full parallelism of the model to be exploited. This is achieved by marking a loop to be unrolled so that the tool will create the implementation with the highest possible degree of parallelism.

```
...
__attribute__((opencl_unroll_hint))
for (unsigned i = 0; i < N; ++i) {
    A[i] = B[i];
}
```

However, the usage of the `opencl_unroll_hint` attribute produces an overhead in terms of area proportional to the unroll factor. Furthermore, considering the code above, each iteration requires the access to an element of the buffer A. In order to pursue an unrolling factor of N, the buffer B must be partitioned. Partitioning an array requires additional resources. The SDAccel environment allows two ways to manage a buffer: partition and reshaping. The `xcl_array_partition` attribute implements an array declared within kernel code as multiple physical memories instead of a single physical memory. It supports cyclic, block-based, and complete partitioning on multi-dimensional arrays. The `xcl_array_reshape` attribute combines the effect of `xcl_array_partition`, breaking an array into smaller arrays, and concatenating elements of arrays by increasing bit-widths.

For instance, in order to get an unrolling factor of N in the code above, buffer A must be fully partitioned, as the snippet below shows.

```
...
int A[BUFFER_SIZE] __attribute__((xcl_array_reshape(complete, 0))
__attribute__((opencl_unroll_hint))
```

```
for (unsigned i = 0; i < N; ++i) {
    A[i] = B[i];
}
```

Loop pipelining. Pipelining a loop results in higher latency and increased throughput, potentially improving the final kernel performance. In fact, although unrolling loops increases concurrency, it does not address the issue of keeping all elements in a kernel data path busy at all times. Even in an unrolled case, loop control dependencies can lead to sequential behavior. The sequential behavior of operations results in idle hardware and a loss of performance. In order to pipeline a loop, the usage of `xcl_pipeline_loop` is required. Consider the following snippet of code, in which each iteration depends on the previous one. The usage of the unrolling in this case leads to a sequential behaviour. An improvement of performance can be obtained by using the `xcl_pipeline_loop` attribute. In this specific case, the compiler infers a shift register.

```
...
__attribute__((xcl_pipeline_loop))
for (unsigned i = 1; i < N-1; ++i) {
    A[i] = A[i-1];
}
```

Task Parallelism. Task parallelism allows the programmer to take advantage of data flow parallelism. In contrast to loop parallelism, when task parallelism is deployed, full execution units (tasks) are allowed to operate in parallel taking advantage of extra buffering introduced between the tasks.

The `xcl_dataflow` attribute enables task-level pipelining, allowing functions and loops to overlap in their operation, increasing the concurrency of the implementation and hence the overall throughput of the design.

The general pattern suggested for use in RECIPE relies on splitting the kernel design into three sub-functions, allowing a dataflow execution:

- An input stage, that handles the input-communication to off-chip memory banks;
- A compute stage, that encapsulates the computational logic fo the acceleration function;
- An output stage, that handles the output-communication to off-chip memory banks;

In this case, the usage of the `xcl_dataflow` attribute allows a concurrent execution of all three stages.

```
__attribute__((reqd_work_group_size (1,1,1))
__attribute__((xcl_dataflow))
__kernel AIO_Stencil ( __global restrict float16 *in,
                         ...
                         ) {

                              ...
                              read_input(..);
```

```
            compute(..);
            write_output(..);
        }
```

OpenCL natively supports a solution to handle inter-kernel communication through *pipes*. A pipe stores data organized as a FIFO. Pipes can be used to stream data from one kernel to another inside the FPGA device without having to use the external memory, which greatly improves the overall system latency. As a consequence, pipes can be used as an alternative to the `xcl_dataflow` attribute. Pipes must be statically defined outside of all kernel functions. The depth of a pipe must be specified by using the `xcl_reqd_pipe_depth` attribute in the pipe declaration.

By using pipes, the kernel structure discussed above requires the definition of three kernels, as shown below.

```
pipe TYPE* input_to_compute;
pipe TYPE* compute_to_output;

__attribute__((reqd_work_group_size (1,1,1))
__kernel read_input ( ... ) {

                        }

__attribute__((reqd_work_group_size (1,1,1))
__kernel compute ( ... ) {

                        }
__attribute__((reqd_work_group_size (1,1,1))
__kernel write_output ( ... ) {

                        }
```

As a result, both the techniques enables task-level parallelism. It is recommended to use pipes when you want depth control over the stream of data between kernel, while it is preferable to use the `xcl_dataflow` attribute to fully rely on compiler optimizations for a task-parallel execution.

4 Conclusions

In this Deliverable we reported on the programming models considered for the implementation of the application use cases in RECIPE. The Deliverable introduced the Abstract Execution Model and the MANGO API, both characterized by the integration of the application execution and lifecycle with the resource management actions. The Deliverable also provided details of the mangolibs runtime support library, which was introduced by the MANGO project and implemented as a set of open source C++ libraries, with additional C bindings. We also described a higher level interface with respect to the mangolibs C/C++ API, in the form of Python bindings for mangolibs, effectively providing a form of Domain Specific Language (DSL) in RECIPE, used to quickly explore possible API extensions as well as higher-level application-specific constructs. As an example, the Python bindings will allow users to easily manipulate an application's recipe from within the application itself, which may be useful to allow a more flexible definition of operating points to cope with unpredictable operating conditions. Finally, the Deliverable covered low-level accelerator programming in RECIPE, targeting OpenCL for both software-programmable and custom HLS accelerators.

References

[1] Giovanni Agosta, Alexandre Dray, José Flich, Edoardo Fusella, Giuseppe Massari, and Marina Zapater. D2.4 Report on WP2 Progress. Technical report, H2020 MANGO Project, 2017.

[2] Giovanni Agosta, Davide Zoni, Giuseppe Massari, Simone Libutti, José Flich, David Atienza, Marina Zapater, Arman Iranfar, Mario Kovac, Josip Knezovic, V. Sruk, and Alessandro Cilardo. D1.3 Report on MANGO Software Support. Technical report, H2020 MANGO Project, 2016.

[3] P. Bellasi, G. Massari, and W. Fornaciari. A RTRM proposal for multi/many-core platforms and reconfigurable applications. In *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–8, July 2012.

[4] José Flich, Giovanni Agosta, Philipp Ampletzer, David Atienza Alonso, Carlo Brandolese, Etienne Cappe, Alessandro Cilardo, Leon Dragić, Alexandre Dray, Alen Duspara, et al. Exploring Manycore Architectures for Next-Generation HPC Systems through the MANGO Approach. *Microprocessors and Microsystems*, 2018.

[5] G. Massari, E. Paone, P. Bellasi, G. Palermo, V. Zaccaria, W. Fornaciari, and C. Silvano. Combining application adaptivity and system-wide resource management on multi-core platforms. In *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, pages 26–33, July 2014.