DEEPHEALTH

D1.2 HPC infrastructure and application adaptation requirements

Project ref. no.	H2020-ICT-11-2018-2019 GA No. 825111
Project title	Deep-Learning and HPC to Boost Biomedical Applications for Health
Duration of the project	1-01-2019 – 31-12-2021 (36 months)
WP/Task:	WP1/ T1.3, T1.8
Dissemination level:	PUBLIC
Document due Date:	30/06/2019 (M6)
Actual date of delivery	30/06/2019 (M6)
Leader of this deliverable	BSC
Author (s)	Eduardo Quiñones (BSC)
Contributors	Jose Flich, Jon Ander (UPV); Thibaut Goetghebuer, François Galea (CEA); Marina Zapater (EPFL); Barbara Cantalupo (UNITO); Tatiana Silva (TREE); Heiko Mauersberger (PRODESIGN); Lluc Alvarez (BSC); Monica Caballero (EVERIS)
Version	V0.10

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 82511

Document history

Version	Date	Document history/approvals
0.1	10/06/2019	First draft contents, including BSC contributions
0.2	16/06/2019	Contributions from UNITO
0.3	18/06/2019	Contributions from TREE
0.4	19/06/2019	Contributions from UPV
0.5	19/06/2019	Contributions from PROD
0.6	19/06/2019	Contributions from EPFL
0.7	24/06/2019	Contributions from CEA
0.8	27/06/2019	SIVECO internal peer review
0.9	28/06/2019	Project Manager (EVERIS) and technical manager (UPV) review
0.10	28/06/2019	Document ready for submission

DISCLAIMER

This document reflects only the author's views and the European Community is not responsible for any use that may be made of the information it contains.

Copyright

© Copyright 2019 the DEEPHEALTH Consortium

This work is licensed under the Creative Commons License “BY-NC-SA”.

Table of contents

DOCUMENT HISTORY **2**

TABLE OF CONTENTS **3**

1 EXECUTIVE SUMMARY **4**

2 THE DEEPHEALTH COMPUTING INFRASTRUCTURE **4**

3 THE API LAYER FOR ECVL AND EDDLL DEVELOPMENT **5**

3.1 PARALLEL PROGRAMMING MODELS 5
3.2 DISTRIBUTED PROGRAMMING MODELS 6
3.3 NON-FUNCTIONAL REQUIREMENTS DESCRIPTION 6
3.4 N2D2 FRAMEWORK 7
3.5 CLOUD-BASED API 7

4 THE SOFTWARE ARCHITECTURE LAYER **7**

4.1 SOFTWARE COMPONENTS 7
4.2 COMPSS 8
4.2.1 COMPSs Runtime Internals 8
4.2.2 Interface with the Underlying Computing Devices 10
4.3 GLOBAL RESOURCE MANAGER 10
4.3.1 The Slurm resource management tool 11
4.3.2 The GRM Allocator 12
4.4 PARALLEL RUN-TIMES 12
4.5 FPGA RUN-TIME 13
4.5.1 Xilinx run-time 13
4.5.2 MANGO run-time 14
4.6 NETLIST PARTITIONING AND VIVADO 17
4.7 OPENSTACK 18

5 THE HW ARCHITECTURE LAYER **19**

5.1 HPC COMPUTING RESOURCES 19
5.1.1 BSC Computing Resources 19
5.1.2 UNITO Computing Resources 20
5.1.3 UPV Computing Resources 20
5.1.4 PRODESIGN Computing Resources 22
5.2 CLOUD-BASED COMPUTING RESOURCES 22
5.2.1 TREE Computing Resources 22
5.2.2 UNITO Computing Resources 22

6 CONCLUSIONS **23**

7 BIBLIOGRAPHY **23**

1 Executive summary

This deliverable covers the work done during 6 months in Task 1.3 "*HPC system definition and capabilities*" and in Task 1.8 "*EDDLL/ECVL requisites for efficient HPC and Cloud adaptation*".

This document describes the computing infrastructure to be developed within the DeepHealth project and upon which the ECV and EDDL libraries will be developed and execute. The *DeepHealth computing infrastructure* will incorporate two type of resources: (1) high performance computing (HPC) resources, including advanced parallel and heterogeneous processor architectures featuring GPUs and FPGAs upon which specific kernels will be accelerated; and (2) big-data cloud based resources. This document also describes the programming models and access methods available by the EDDLL and ECVL libraries for an efficient exploitation of the performance capabilities of the DeepHealth computing infrastructure.

Tasks 1.3 and 1.8 have been carried out successfully and the related project objectives have been reached and documented in this deliverable.

2 The DeepHealth Computing Infrastructure

Figure 1 provides the block diagram of the *computing infrastructure* proposed by the DeepHealth project. The infrastructure will be composed of three main layers: (1) an abstraction programming interface (API) exposed to ECVL and EDDLL developers to exploit the performance capabilities of the underlying computing resources; (2) the software (SW) architecture composed of a set of run-time frameworks in charge of efficiently managing the parallel and heterogeneous execution of the HPC and cloud-based computing resources; and (3) a set of hardware (HW) computing resources. The Figure also identifies for each partner the set of computing resources offered to the project.

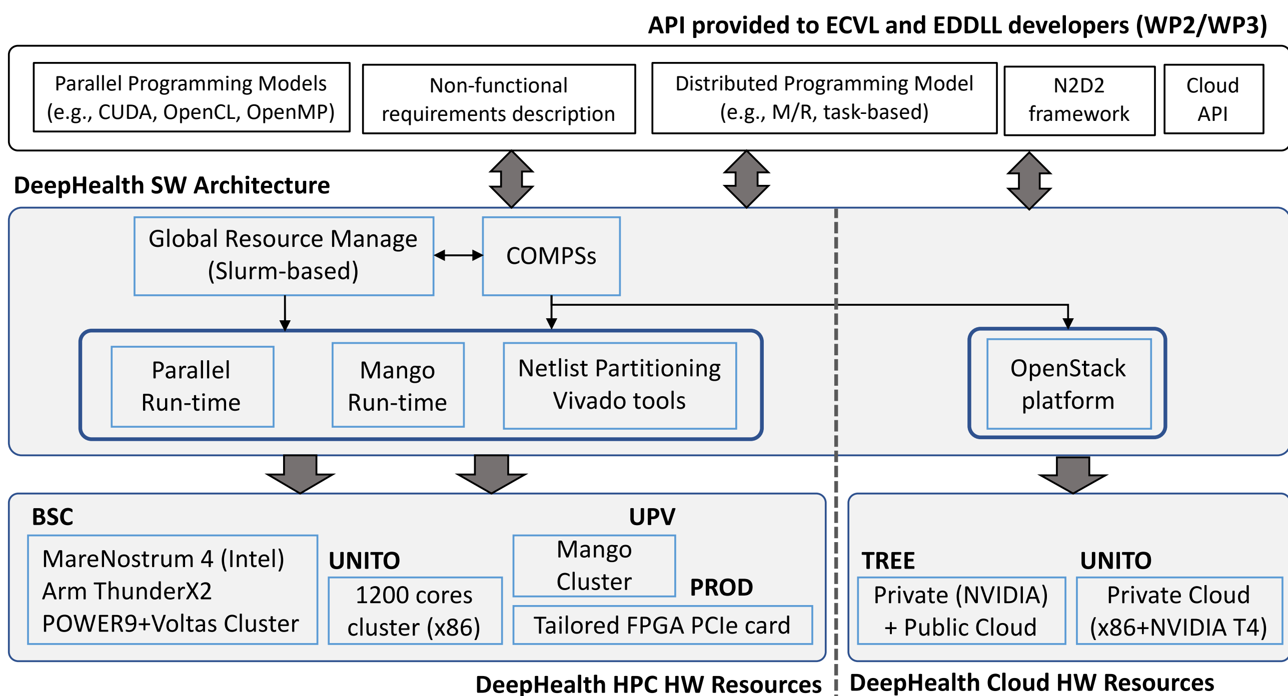

Figure 1. Block diagram of the DeepHealth computing infrastructure.

Next subsections summarizes each of the components that form the DeepHealth computing infrastructure.

3 The API Layer for ECVL and EDDLL Development

This section describes the API that DeepHealth will expose to the ECVL and EDDLL developers to efficiently exploit the parallel and heterogeneous HPC and cloud-based capabilities of the DeepHealth computing resources. The success of the exposed API will rely on its *productivity*, which combines *programmability*, *portability* and *performance*:

1. *Programmability* refers to the capability of the API to provide the right level of abstraction to identify the entities relevant of the program for its execution, e.g., units of parallelism, synchronization, data dependencies and data transfers, while hiding the complexities of the underlying platform.
2. *Portability* refers to the property of the API to collect the required information, so the run-time is able to execute the same program in different computing platforms, achieving the maximum (possible) performance of the program executed on a given platform.
3. *Performance* refers to the property of maximising the exploitation of the parallel and heterogeneous capabilities of the program and the underlying platform.

3.1 Parallel Programming Models

Parallel programming models are of paramount importance to develop parallel applications, while hiding the processor complexities. They can be classified in three main groups: (1) those supporting only homogeneous and shared memory execution model; (2) those specialised on heterogeneous and distributed memory execution model featuring acceleration devices such as GPUs, many-core fabrics or SoC-FPGAs; and (3) those supporting both execution models, homogeneous and heterogeneous. Due to the heterogeneous nature of the computing resources available in the project, DeepHealth will focus only on the second and the third group:

- In the group of parallel programming models specialised on *heterogeneous computing*, **OpenCL** [1] and **CUDA** (Compute Unified Device Architecture) [2] are currently the dominant standards for parallel programming models specialised on acceleration devices inspired on GPU programming. OpenCL is an open-source standard maintained by the Khronos Group, while CUDA is property of the GPU vendor NVIDIA. Both are very similar in terms of execution model as both define a platform model upon which the parallel program executes. The unit of parallelism are kernels, which are grouped in blocks, and these in grids, creating a software hierarchy that requires to match with the memory hierarchy of the underlying accelerator for better performance. **OpenACC** [3] is a higher-level abstraction task-based programming model that offers syntax to off-load computations to hardware accelerators with a simple, yet powerful data movement interface between the host and the accelerators' memory(ies).

- In the group of parallel programming models specialised on both *homogeneous and heterogeneous computing*, **OpenMP** [4], traditionally the de-facto standard for shared-memory programming in HPC, implements a powerful tasking model that allows expressing fine-grained and unstructured parallelism augmented with features to express data dependencies. The tasking model is coupled with the acceleration model to facilitate the coordination between the host and acceleration devices, providing support for synchronization and communication among them. OmpSs [5], the programming model developed by the partner BSC and upon which the OpenMP tasking model was inspired, supports a very similar tasking model, extending the acceleration model by enabling to offload acceleration kernels from other programming models, such as CUDA, OpenCL or FPGA bit-streams.

Moreover, with the objective of supporting the approach followed by EDDL, in which the control program manages which devices to be used (CPU, GPU, FPGAs), the parallel programming model will allow the identification and isolation of components and algorithms to be deployed on different architectures (e.g. convolutions, matrix multiplications, activation functions). A list of identified kernels and functions can be found in the Deliverable *D1.3. API specifications for EDDLL and ECVL libraries*.

3.2 Distributed Programming Models

DeepHealth will consider distributed computing programming models supporting Map Reduce (M/R), RDD and tasking programming models, very well-known models for the development of HPC and big data applications:

- M/R is a programming model for processing parallelizable problems across large datasets using a large number of nodes. Processing can occur on data stored either in a filesystem or in a database. M/R can take advantage of the locality of data, processing it near the place it is stored in order to minimize communication overhead. This programming models are supported by Hadoop [6], Spark [7] and COMPSs [8] respectively.
- The task-based programming model is based on the sequential development of program in which the user is mainly responsible for identifying the functions to be executed as asynchronous parallel tasks and the data dependencies existing among them. The runtime system is then in charge of exploiting the inherent concurrency of the code, automatically detecting and enforcing the data dependencies between tasks and spawning these tasks to the available resources. This programming model is supported by COMPSs.

DeepHealth will pay special attention to the COMPSs distributed framework developed (and owned) by the partner of this project BSC, as it provides an unify programming environment to exploit both the structured parallelism supported by M/R model, and unstructured parallelisms supported by the task-based model (see Section 4.2 for further details).

3.3 Non-Functional Requirements Description

The DeepHealth project aims at achieving two goals which are directly related to the platform infrastructure deployment, namely, the reduction of the time-to-model-in-production (ttmip), and the efficient and transparent use of the heterogeneous resources to achieve the highest performance/power/efficiency metrics. We tackle these challenges by proposing a two-fold approach:

- The development of a framework that enables expert users to launch their application (i.e., training workflows) on top of the underlying heterogeneous HPC infrastructure in a transparent way. This will be done by using the Global Resource Manager (GRM) of EPFL, which will serve as a single entry-point for all applications. We will extend the GRM to tackle the specific allocation challenges of deep learning training and integrating it with the COMPS runtime.
- The development of multi-objective heterogeneity-aware workload allocation policies that, incorporated into the GRM, will enable to allocate the different training requests to the most adequate resources. These policies will be based on machine-learning techniques (and more specifically reinforcement learning) and will take into consideration both the time-of-training-models (totm) and the time-of-preprocessing-images (topi) when such a process can be done in an automated way.

The development of the multi-objective workload allocation policies requires an initial step of profiling the workloads on DeepHealth to understand their stress on the underlying system and understand the set of resources that would potentially be needed depending on the input and accuracy requirements. In this regard, once the different kernels required by the use cases are identified, we will study their most likely combinations, and also the relevant pre-processing stages required on those kernels. This creates a set of most probable workflows. Once this is defined for each of the different applications, we will assess their computational and memory requirements. This will lead us to a high-level understanding of their intensity that will help us to guide the process of runtime allocation of computation to the heterogeneous resources.

3.4 N2D2 Framework

N2D2 (Neural Network Design & Deployment) is a neural network framework developed by CEA. The framework provides a fully integrated solution allowing simple and fast exploration of different network topologies to develop neural network-based applications.

The platform supports data pre-processing and augmentation, neural network topology design, benchmarking of both performance and resources usage, network quantization to reduce its size and inference support for different hardware platforms, all in one unified framework. From one network trained on GPU through N2D2, it's easy to export the network for inference to a GPU (with OpenCL, Cuda or TensorRT), a CPU, a microcontroller or a FPGA depending on the needs.

During the DeepHealth project, N2D2 will be improved to support the usage of multiple GPU during the neural network training phase. This will allow N2D2 to train large and deep neural networks faster than with one GPU.

3.5 Cloud-based API

Once an EDDLL or ECVL application will be built, the COMPSs runtime will handle how the workload is distributed in the targeted cloud infrastructure. The COMPSs runtime has two configuration files for this aim: *resources.xml* and *project.xml*. These files contain information about the execution environment and are completely independent from the application. The decision of how (and where) the application will execute will be configured properly in these files.

The COMPSs runtime communicates with a cloud manager by means of connectors. Each connector implements the interaction of the runtime with a given provider's API. In the specific case of a deployment in the TREE hybrid cloud solution, two connectors can be used with this purpose:

- OCCI: It could be interesting to use cloud resources in an IaaS way. In addition to that, OpenStack is compatible with the standard.
- Mesos: It will be the best option if a PaaS way is preferred.

4 The Software Architecture Layer

The software architecture layer incorporates all the runtime frameworks needed to: (1) distribute the computation across the multiple computing nodes available at the hardware level, and (2) orchestrate the parallel and heterogeneous computation within the computing nodes. This layer will support all the parallel and distributed programming models, as well as the different API, offered to ECVL and EDDLL developers at the API layer.

4.1 Software Components

DeepHealth has carefully selected the software components that will form the software development ecosystem upon which the EDDLL and the ECVL will be developed, prioritizing those owned by the DeepHealth partners or offered as open-source with a large community behind. By doing so, we envision to reduce the time-to-market and maximize exploitation opportunities of the DeepHealth computing infrastructure. Table 1 identifies the set of software components that will be included in the software architecture layer, the owner and the license. In the next subsections, the software components are described in detail.

Table 1. Software components aimed to be included in the DeepHealth computing infrastructure.

SW Tool	Owner	License
COMPSs	BSC	Open-source
Global Resource Manager	EPFL	Open-source
OpenMP	OpenMP ARB	Open-source
CUDA	NVIDIA	Proprietary
OpenCL	Khronos Group	Open-source

Mango run-time	UPV	Open-source
Netlist partitioning & Vivado Tools	Xillinx	Proprietary
OpenStack	OpenStack Foundation	Open-source
N2D2 Framework	CEA	Proprietary
Docker	Docker Inc.	Proprietary

4.2 COMPSs

COMPSs, developed and owned by the DeepHealth partner BSC, provides a complete framework, composed by a programming model (described in Section 3) and a runtime system, which is the focus of this section, enabling the development of parallel applications for distributed infrastructures. COMPSs enables the software development at a very low cost because of two main reasons: (1) the model is based on sequential programming on top of popular programming languages (i.e., Java, Python and C/C++), meaning that users do not have to deal with the typical duties of parallelization and distribution (e.g., thread creation, data distribution, fault tolerance, etc.); and (2) the model abstracts the application from the underlying distributed infrastructure, hence COMPSs programs do not include any detail that could tie them to a particular platform (e.g., deployment or resource manager) boosting portability among diverse computing infrastructures.

4.2.1 COMPSs Runtime Internals

The COMPSs runtime is organised in a master-worker structure as depicted in Figure 2:

- The *master* part executes in the resource where the application is launched, i.e., where the main program runs, and is responsible for steering the distribution of the application, as well as for implementing most of the features of the runtime concerning task processing and data management and movement.
- The *worker* side is in charge of responding to task requests coming from the master, although in some designs such as clusters it also has data transfer capabilities, and it can be transient (i.e., a new runtime process is started every time a task request arrives) or persistent (i.e., a process remains in the resource all along the application lifetime).

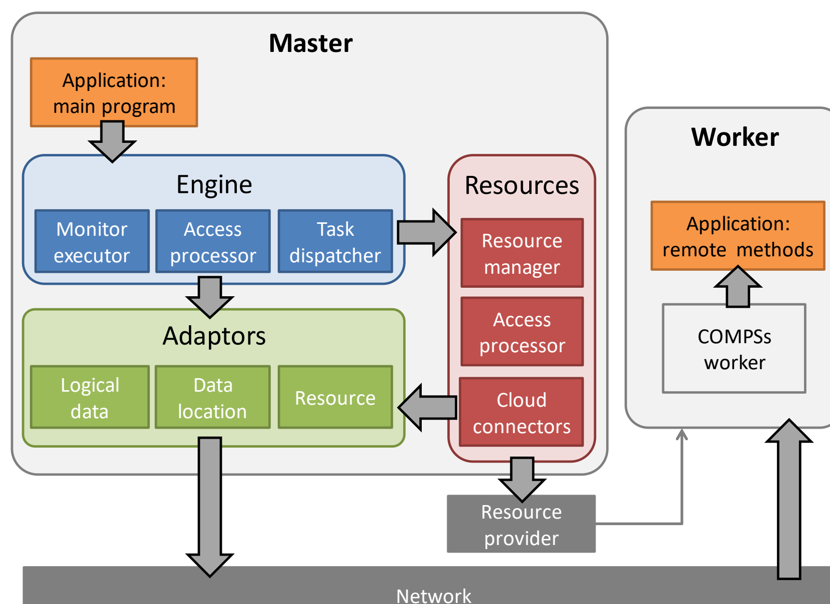

Figure 2. COMPSs runtime internals overview

The execution of a COMPSs program is summarized in Figure 3, and it involves the following stages:

1. *Instrumentation* of the main program. The first phase of the execution of a COMPSs application consists of two parts: first, the methods selected by the programmer are replaced by the asynchronous creation of their associated tasks, and then the data accesses specified for these methods are checked in order to ensure the sequential memory consistency.
2. *Data dependence analysis*. As the main program runs, the runtime receives task creation requests. The data consumed and produced by the task is used by the data dependence analysis mechanism, which dynamically builds a *Task Dependency Graph* whose nodes are tasks, and whose arrows symbolise the dependences. This graph represents the workflow of the application and imposes what can and cannot be run concurrently.
3. *Data renaming*. In order to expose more parallelism in the applications, data causing Write-after-Read (WaR) and Write-after-Write (WaW) dependences is renamed. The runtime keeps track of all data accessed by the application and the versions of this data created after the renaming process; hence it can guarantee the sequential memory consistency of the application.
4. *Task scheduling*. A task remains in the Task Dependency Graph until all its predecessors have completed, hence its dependences are solved. Then, using the list of worker resources the runtime is provided with, if the runtime is able to find an available resource, the task is scheduled. Otherwise, the task is added to a queue of pending tasks waiting for a free resource. Different policies allow mapping tasks to resources (e.g., based on data locality, in a round robin fashion, etc.). Additionally, there is a *pre-scheduling* mechanism offered in the runtime in order to send the data needed by a task before the task can be executed, overlapping computation and communication. This way, when the processor where the task is to be executed gets free, the task can be submitted without waiting for any transfer.
5. *Task submission*, *execution* and *monitoring*. Once a task is ready to be executed, its input data is transferred and the target resource is free, then the master runtime asynchronously submits the task and registers the notifications coming from the worker resource informing about the completion of the task. In the worker resource, the worker part of the runtime is in charge of executing the task. Furthermore, the master runtime implements a fault-tolerant mechanism that allows for retrying the submission either in the same resource or in a different one. Finally, when the task completes, the runtime removes it from the Task Dependency Graph.

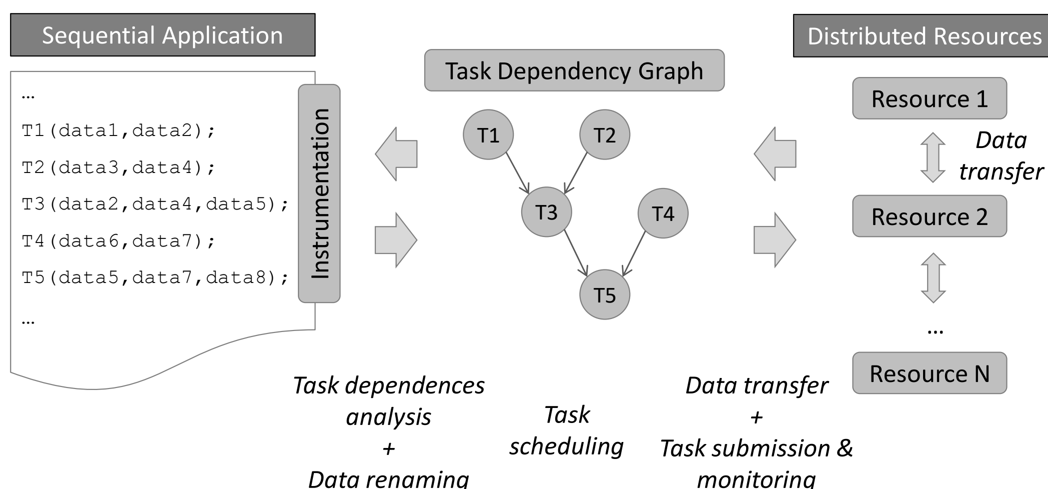

Figure 3. Execution pipeline of a COMPSs application

4.2.2 Interface with the Underlying Computing Devices

An important feature of COMPSs is the capability to execute applications transparently with regards the underlying infrastructure. With such a purpose, COMPSs implements the interaction between the runtime and the computational resources (i.e., physical resources or VMs) by means of different adaptors, each implementing the specific providers APIs. This mechanism makes possible the execution of computational loads on fog environments without the need of adapting the code, hence providing scalability and elasticity properties. Currently, there are two adaptors implemented: (1) Non-blocking I/O, NIO, which offers high performance in secured environments, and (2) GAT, which offers interoperability with diverse kinds of Grid middleware.

Together with the adaptors, the COMPSs runtime uses connectors to communicate with the cloud managers. Each connector implements the interaction of the runtime with a given provider's API, supporting four basic operations: (1) ask for the price of a certain VM in the provider, (2) get the time needed to create a VM, (3) create a new VM and (4) terminate a VM. This design allows connectors to abstract the runtime from the particular API of each provider and facilitates the addition of new connectors for other providers. Currently COMPSs implements four different connectors: (1) JClouds [8], (2) Docker [9], and (3) Mesos [10].

Flexibility is one of the main features of COMPSs. With that in mind, COMPSs has been integrated with several programming models in order to better exploit the capabilities of the different target architectures. This means the COMPSs programmers can define computing elements in several programming languages: (a) OpenCL, for GPGPU programming, (b) OmpSs for CPU, GPU and Cluster programming, or (c) MPI [11], for Cluster programming. The integration between COMPSs and OmpSs is particularly interesting, because OmpSs further integrates other programming languages such as CUDA, OpenCL and MPI. This means that COMPSs not only allows for flexible, programmable and portable programming of both edge and cloud devices, but also supports a performance-aware environment where specific programming models can be used to exploit the special features of each target architecture.

4.3 Global Resource Manager

The goal of the Global Resource Manager (GRM) is to provide a single entry-point to the applications being run on the HPC computing infrastructures in DeepHealth, while at the same time allocating tasks in the most performance and energy-efficient way to the heterogeneous underlying resources. Furthermore, the GRM in DeepHealth will also be able to act as an interface and API to the different platforms deployed in DeepHealth. This API will enable running different workloads directly on the clusters, and retrieving the training results obtained.

From a functionality-wise perspective, the GRM is composed of the following main components:

- The resource manager software, which is in charge of managing incoming workloads, scheduling (i.e., queuing them) and allocating them to the nodes. This function is undertaken by the Slurm resource manager.
- The workload allocation policies (in what follows "GRM Allocator"), which take decisions on the specific allocation of tasks to nodes. The GRM allocator contains the power/performance/thermal- aware policies, which are in charge of improving the efficiency and performance of the system.
- Data manager and data retrieval services that allow interfacing with both COMPS to coordinate workload allocation and the different platform APIs (when needed).

From a purely implementation perspective, the GRM consists on a bunch of services working together in a coordinated way. Therefore, when a new application is launched in the cluster via the entry point, the GRM Allocator will apply one upon the various policies developed ad-hoc for the MANGO runtime (see Section 4.5.2 for further details), developed in the framework of the EU MANGO project[1], which

[1] http://www.mango-project.eu/

will decide the node where the application will be executed, and will finally perform the allocation, by executing the SLURM Controller (in particular the slurmctld), which will, in its turn, using the slurmd of the selected node, assign the task to a node and pass its control to COMPS for the runtime management. This flow is depicted in the next figure.

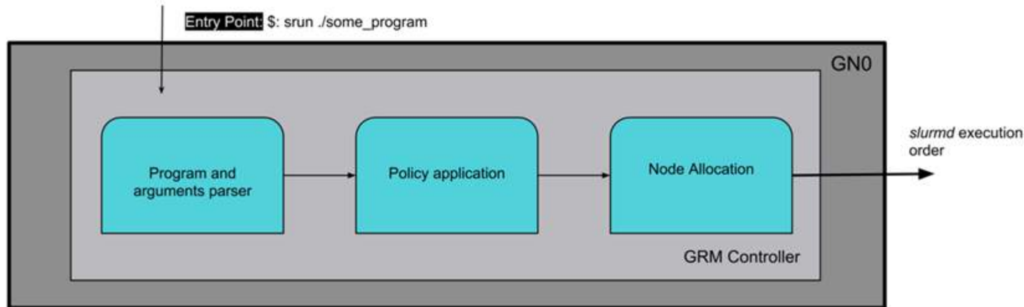

Figure 5. GRM Flow: entry-point to applications, and allocation via SLURM

To facilitate deployment and in order to achieve the objectives of replicability and robustness, the GRM will be running under docker containers. Because of its dockerized deployment, the GRM can be used in virtually any HPC infrastructure which runs a LinuxOS supporting dockers.

4.3.1 The Slurm resource management tool

SLURM was selected to accomplish the task of single entry-point to the system as it is a highly scalable cluster management tool. However, because it is a general-purpose HPC cluster management tool, it does not provide the heterogeneity-awareness required in DeepHealth. This heterogeneity awareness is achieved by means of its integration with the COMPS runtime and, subsequently, with the MANGO runtimes.

SLURM works in a centralized way having a central manager called “slurmctld” (SLURM controller) responsible of monitoring the status of applications and the resource availability. On the other hand, each node will be running a SLURM daemon, or “slurmd”, which supervises applications running on each node and reports its status. In short, SLURM daemons work as a remote shell for the controller. Moreover, all the information gathered by SLURM, can be stored in a MySQL database, managed by the “slurmdbd” daemon, which runs in the controller and records accounting information, historical data and current status of the nodes, among others.

An overview of how SLURM works is depicted in Figure 4 (taken from the official documentation).

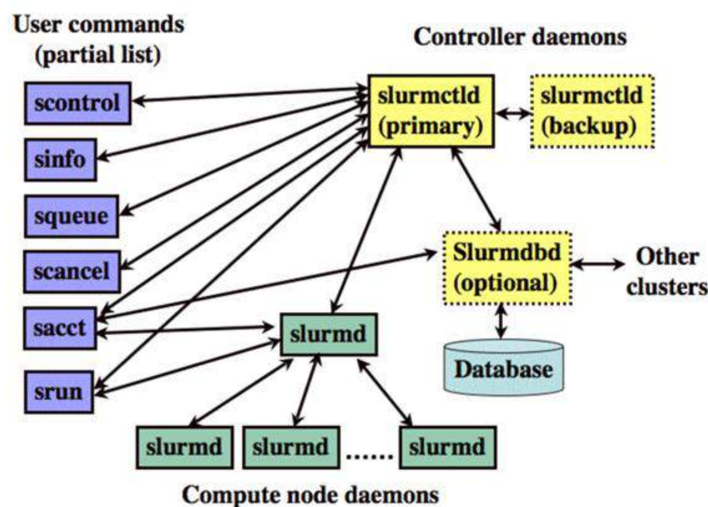

Figure 4. Slurm basic components diagram and overview

4.3.2 The GRM Allocator

The GRM allocator accomplishes the goal of providing power, performance and energy aware management policies. The GRM allocator will implement two different types of policies in DeepHealth.

From an implementation perspective, the GRM Allocator needs to be able to express the dependencies and behaviour of the heterogeneous underlying hardware, abstracting the view from the whole HPC cluster perspective. The GRM Allocator does so by generating a graph network which contains all the available nodes. This support is implemented by using the Networkx Python library. Networkx provides all the necessary tools to manipulate, study and distribute for complex networks graphs. For deciding a specific allocation DiGraph Networkx object (which behaves as a bidirectional graph) is instantiated.

The next figure shows an example graph for the MANGO architecture. In this case we build a network where the main node is the controller of the global resource manager (Called Master). This node is followed by the General Purpose nodes and then, attached to them the Heterogeneous Nodes. In this last layer, unlike the previous where the 64 cores of each node are represented by the same node, each accelerator core will have its own node. Edges have a normalized weight between 0 and 1 that represents the suitability of that specific node for the execution of the workload, with respect to a specific performance/energy objective.

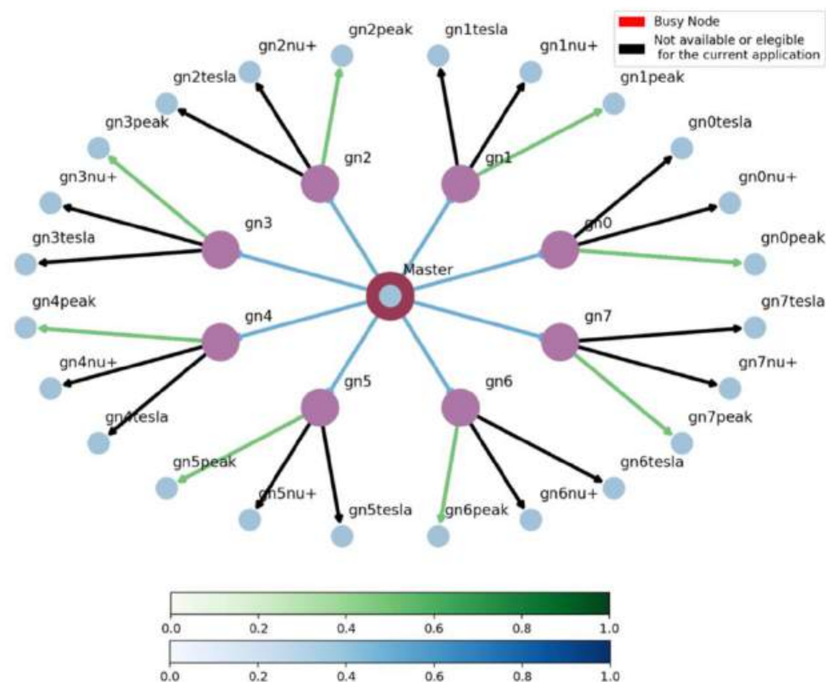

Figure 4. Graph used for current allocation policies in the MANGO cluster. Bodes represent resources and edges are weighted depending on the performance/efficiency of the accelerator for executing a particular task.

The current policies supported by the GRM traverse this graph by assigning different weights to the edges depending on the allocation objectives using heuristics. Within DeepHealth, EPFL will propose drastically new allocation algorithms that will be able to tackle the complexity of deep learning training workloads. These algorithms will be based on reinforcement learning techniques (and specifically Q-learning).

4.4 Parallel Run-times

Current parallel programming models rely on run-time libraries to dynamically assign parallel entities to processor resources with the objective of maximizing performance, while respecting the parallel execution model.

In heterogeneous computing, the offloading of computation to accelerators requires the sharing of code and data between host and accelerator. Two mechanisms are dominant on modern heterogeneous systems: *copy-based*, which requires the copy of code and data to the private accelerator memory space; and *unified virtual memory* (UVM), which enables zero-copy data sharing by supporting virtual memory addressing from the accelerator. The major vendors of high-end heterogeneous systems on chip (SoCs) have products on the market that support the UVM according to the OpenCL or NVIDIA CUDA specification.

Moreover, run-times supporting heterogeneous computing have the capability of (dynamically) deciding whether a parallel kernel is offloaded to the accelerator or executed on the host. This feature, supported by OpenMP and SoC-FPGA controllers run-times, provides a great support to better adapt the heterogeneous computing to execution conditions. To do so, the executable incorporates the two compiled versions of the kernel, i.e. one supported by the host and another supported by the accelerator.

4.5 FPGA Run-time

4.5.1 Xilinx run-time

Xilinx FPGA devices are becoming very popular in the field of machine learning to accelerate neural networks computation. One of the main reasons of the recent uptake of Xilinx devices is the simplicity of its use. Traditionally, programming FPGA devices has been a complex task that required deep knowledge of the underlying hardware and thus, has been generally considered out of the scope of most programmers in HPC and machine learning. However, the recent release of the Xilinx software development acceleration tool (SDA) has made possible developing FPGA applications with low complexity. The Xilinx SDA tool allows the utilization of high-level synthesis (HLS) to implement kernels written in C in the FPGA and the direct use of kernels implemented in a hardware description language (RTL). Developing kernels with C and HLS reduces the complexity of implementing FPGA kernels and requires lower expertise from the programmer side.

The SDA tool implements the Xilinx run-time (XRT) to allow the user to easily offload computations from the host to the FPGA devices. At the host side the most common approach is using OpenCL as the programming language. With OpenCL the user can from the host side define which of the applications kernels will be executed in the FPGA and how the data is transferred from the host to the device and vice versa. The XRT is implemented as a combination of userspace and kernel driver components supporting PCIe based accelerator cards and provides standardized software interface to Xilinx FPGA such as the OpenCL. The key user APIs are defined in xclhal2.h header file. Figure 4, taken from the XRT public documentation[2], illustrates the XRT software architecture.

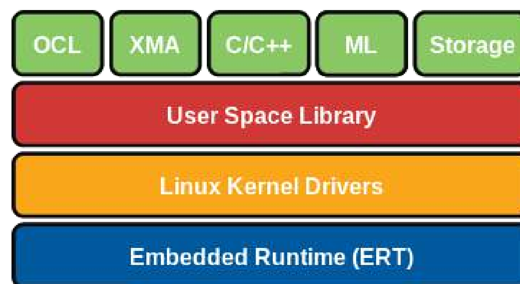

Figure 4. XRT stack (extracted from the XRT public documentation).

In the context of DeepHealth we will focus on the Xilinx platforms that can be programmed using the SDA tool. The FPGA device in these platforms is programmed with a static *shell* and a reconfigurable (dynamic) region. The static region covers all the circuit logic required to provide the interface to the host and memory whereas the dynamic region is the one that is used to include the application specific

[2] https://xilinx.github.io/XRT/2018.3/html/index.html

kernels. The reconfigurable region contents are compiled by the user using SDA compiler tool chain to produce an FPGA binary file (aka *xclbin).*

4.5.2 MANGO run-time

One of the target infrastructures in the project is the MANGO prototype, developed in the framework of the EU MANGO project. It consists of a set of FPGA clusters interconnected to HPC servers via PCIe connection. In the MANGO project a complete run-time solution was deployed in order to allow the deployment of applications running both at the servers nodes and at the FPGA clusters. Figure 5 shows the basic context addressed by the MANGO run-time. We can see a general-purpose node (GN) with a multi-FPGA cluster attached via PCI express. The FPGA includes some accelerators (PEAK and NUP units) and some memories (DDR3 or DDR4).

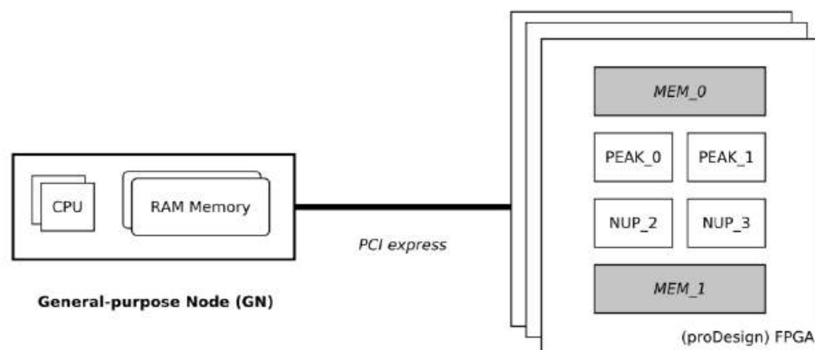

Figure 5. MANGO context (one GN server and one FPGA cluster).

The MANGO runtime is decomposed in a library and a daemon process. The API (HN library) allows an application to communicate with running kernels on the FPGAs from the server. It allows for efficient communication by writing and reading into allocated buffers in DDR memories spread over the FPGA cluster. Synchronization primitives are defined as well to enable kernels on the FPGAs and applications on the servers to collaborate and run concurrently. Multiple transfer modes can be used in the HN API as well as monitoring capabilities for the underlying network deployed in the clusters and the accelerator units defined within the FPGAs. Figure 6 shows how HN daemon and libraries interplay in a setting with different applications running on the system (BBQE is a resource manager).

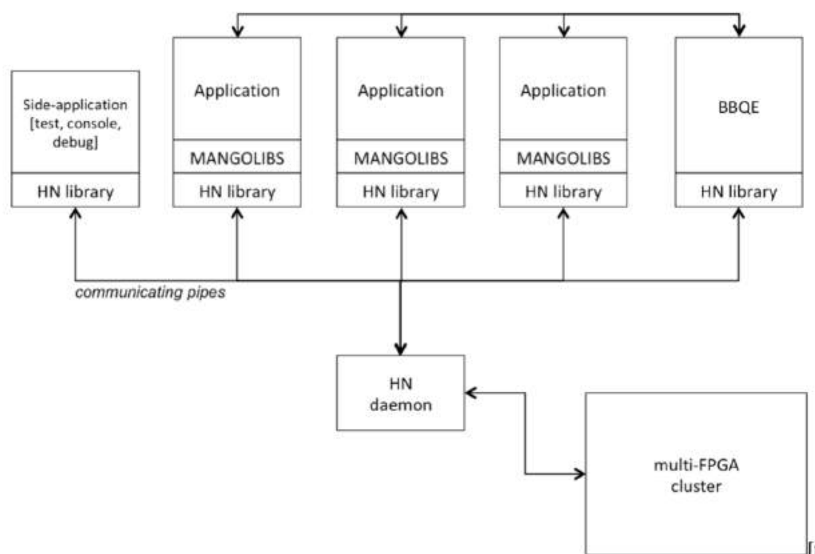

Figure 6. MANGO HN daemon and libraries.

The running process (HN daemon) allows multiple applications running at the same time, each one targeting different accelerator units within the FPGA clusters and concurrently transferring data at the same time. Figure 7 shows the schematic of the current HN daemon.

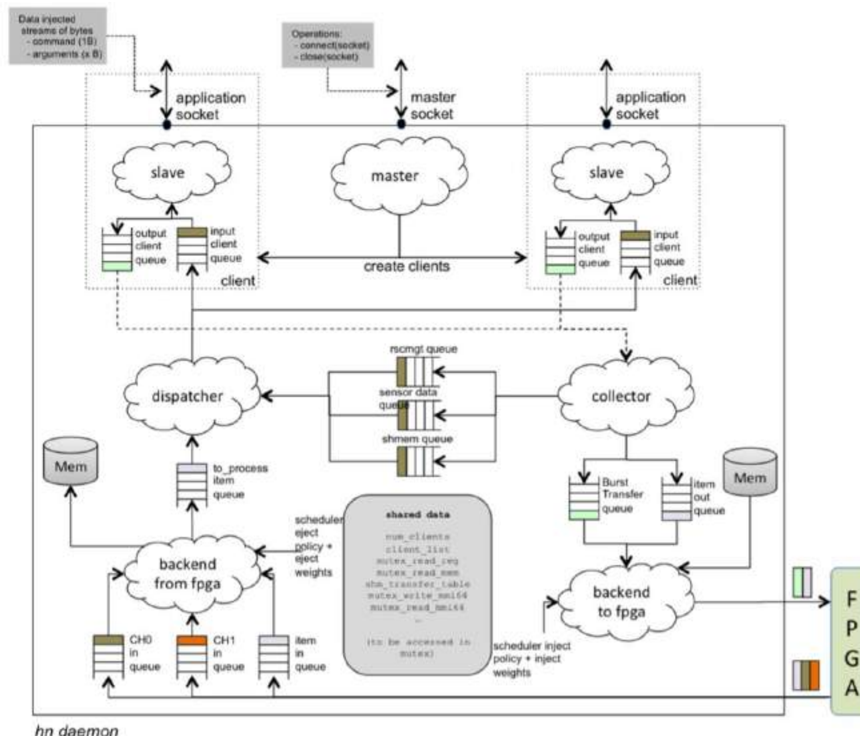

Figure 7 MANGO HN daemon.

The MANGO runtime can be adapted also to multiple configurations of the FPGA clusters, enabling a custom adaptation of the hardware (number and type of FPGAs, DDR memories, interconnects) to the specificities of the target application.

Another API (MANGO API) has been deployed to enable applications to transparently request resources from the clusters and manage them in a unified and simplified way. Indeed, a resource manager (BBQE) is added to the MANGO solution taking care of resource usage.

At the end, the MANGO API and its associated run-time is similar to the OpenCL API and run-time. In the next sections we will show the main API functions.

In DeepHealth, the complete run-time of MANGO will be put at the service of the libraries deployed (both EDDL and ECVL).

4.5.2.1 HN API definition (part I)

API function	Used by	Description
		Boot, initialization and termination
hn_initialize	Both	Initializes communication with HN daemon
hn_end	Both	Ends communication with HN daemon
hn_load_architecture	RM	Loads an architecture into the FPGA cluster
hn_reset	RM	Resets FPGA cluster
		Configuration information access
hn_get_num_tiles	RM	Provides the number of tiles in the cluster
hn_get_num_vns	RM	Provides the number of virtual networks
hn_get_tile_info	RM	Delivers tile configuration information
		Statistics and Sensors
hn_get_tile_temperature	RM	Provides the temperature associated with a tile
hn_get_tile_stats	RM	Gets statistics related to a tile
		Configuration process
hn_boot_unit	APP	Boots an accelerator unit
hn_write_image_into_memory	APP	Writes a memory image into a memory
hn_load_kernel	APP	Loads a kernel into memory
hn_run_kernel	APP	Runs a kernel in a unit
hn_set_tlb	APP	Configures a TLB entry associated with a unit
		TILEREG register access
hn_read_register	Both	Reads a TILEREG register value
hn_write_register	Both	Writes a TILEREG register content
		Memory access and DMA transfers
hn_read_memory	APP	Reads from a memory buffer located in a DDR
hn_write_memory	APP	Writes to a memory buffer located in a DDR
hn_shm_malloc	APP	Allocates a buffer for shared memory access
hn_shm_free	APP	Frees a shared memory access buffer
hn_shm_write_memory	APP	Writes to a buffer in a DDR from shared memory
hn_shm_read_memory	APP	Reads from a buffer in a DDR to a shared memory
hn_register_dma_operation	APP	Registers a DMA transfer associated to a DDR
hn_dma_to_unit	APP	Triggers a DMA operation from a DDR to a unit
hn_dma_to_mem	APP	Triggers a DMA operation from a DDR to another DDR
hn_release_dma_operation	APP	Releases a DMA operation
hn_get_dma_status	APP	Gets DMA transfer status
		Unit's specific functions (PEAK)
hn_peak_reset	Int	Resets a PEAK unit
hn_peak_get_console_char	Int	Gets console from a PEAK unit
hn_peak_send_char	Int	Sends characters to a PEAK unit console
hn_peak_echo_enable	Int	Enables ECHO for the PEAK unit
hn_peak_echo_disable	Int	Disables ECHO for the PEAK unit
hn_peak_get_utilization	Int	Gets PEAK utilization info

4.5.2.2 HN API definition (part II)

API function	Used by	Description
Synchronization		
hn_get_synch_id	APP	Gets a synchronization ID
hn_get_synch_id_array	APP	Gets an array of synchronization IDs
hn_release_synch_id	APP	Releases a synchronization ID
hn_release_synch_id_array	APP	Releases an array of synchronization IDs
hn_read_synch_register	APP	Reads a synchronization register via its ID
hn_write_synch_register	APP	Writes a synchronization register via its ID
Interrupts		
hn_register_int	APP	Registers an interrupt handler
hn_interrupt	APP	Triggers an interrupt to a unit
hn_wait_int	APP	Waits an interrupt from a unit
hn_release_int	APP	Releases an interrupt handler
Resource management support		
hn_lock_resources_access	RM	Locks resource management functions access
hn_unlock_resources_access	RM	Unlocks resource management functions access
hn_find_memory	RM	Finds a memory buffer
hn_find_memories	RM	Finds a set of memory buffers
hn_allocate_memory	RM	Allocates a memory buffer
hn_release_memory	RM	Releases a memory buffer
hn_get_available_network_bandwidth	RM	Provides available network bandwidth
hn_get_available_read_memory_bandwidth	RM	Provides available read memory bandwidth
hn_get_available_write_memory_bandwidth	RM	Provides available write memory bandwidth
hn_get_available_read_cluster_bandwidth	RM	Provides read cluster access bandwidth
hn_get_available_write_cluster_bandwidth	RM	Provides write cluster access bandwidth
hn_reserve_network_bandwidth	RM	Reserves network bandwidth
hn_reserve_read_memory_bandwidth	RM	Reserves read memory bandwidth
hn_reserve_write_memory_bandwidth	RM	Reserves write memory bandwidth
hn_reserve_read_cluster_bandwidth	RM	Reserves read cluster access bandwidth
hn_reserve_write_cluster_bandwidth	RM	Reserves write cluster access bandwidth
hn_release_network_bandwidth	RM	Releases network bandwidth
hn_release_read_memory_bandwidth	RM	Releases read memory bandwidth
hn_release_write_memory_bandwidth	RM	Releases write memory bandwidth
hn_release_read_cluster_bandwidth	RM	Releases read cluster access bandwidth
hn_release_write_cluster_bandwidth	RM	Releases write cluster access bandwidth
hn_find_units_set	RM	Finds a set of units
hn_find_units_sets	RM	Finds all posible sets of units
hn_reserve_units_set	RM	Reserves a set of units
hn_release_units_set	RM	Releases a set of units
Error handling		
hn_print_error	Any	Prints through console last error occurred

4.6 Netlist Partitioning and Vivado

When designing applications for a FPGA target, at the lowest abstraction level the application is represented as a network of interconnected components. Those components are instances of the resources that are available on the FPGA (flip-flops, lookup tables, DSP, BRAM, ...), interconnected with *nets* (or wires) to form a specialized electronic circuit design. The description of this network is called a *netlist*. Typically, that netlist would be placed and routed on a description of the FPGA hardware, resulting in a bitstream file to be used to configure the FPGA.

Netlists are the result of the conversion from a higher-level representation of the application written in a Register-Transfer Level (RTL) language, typically VHDL or Verilog. This RTL to netlist conversion is called the synthesis phase.

All those phases (synthesis, place and route, and bitstream generation) are present in the Xilinx development environment Vivado. However, Vivado projects typically target only one FPGA.

The N2D2 framework developed at CEA allows the generation of specialized circuits for deep neural networks (see Section 3.4), according to a high-level description of the DNN. This FPGA-targeted DNN technology, also designed at CEA, is called DNeuro. Its output comes in the form of an RTL description of the neural network. Like any FPGA project, this RTL description can be imported in Vivado to be synthesized, placed and routed, to be implemented on the desired FPGA hardware.

In DeepHealth, we will work on partitioning large netlists that cannot fit in a single FPGA, so that they are deployed on a multi-FPGA platform consisting in tightly-coupled FPGAs. Ideally, the developed technology will act as an abstraction of a very large FPGA, so the original design does not have to be modified. The partitioner will take as input the netlist to be partitioned and a description of the target platform (including the detail of communication links between I/O ports of the different FPGAs), and will output one netlist for each of the FPGAs in the platform, to be placed and routed in Vivado. The goal is to be able to deploy very large DNeuro networks on multi-FPGA platforms.

4.7 OpenStack

The OpenStack project [13] is an open source cloud computing platform that supports several types of cloud environments. It is a set of software tools for building and managing cloud computing platforms for public and private clouds. The project aims for simple implementation, massive scalability, and a rich set of features.

OpenStack lets users deploy virtual machines and other instances that handle different tasks for managing a cloud environment on the fly. It aims horizontal scalability for a big variety of projects. Besides that, it is possible not only using on-premise hosts, it is very usual to use public cloud providers to make bigger our private cloud.

OpenStack is made up of many different moving parts. Because of its open nature, anyone can add additional components to OpenStack to help it to meet their needs. But the OpenStack community has collaboratively identified nine key components that are a part of the "core" of OpenStack, which are distributed as a part of any OpenStack system and officially maintained by the OpenStack community:

- **Nova** is the primary computing engine behind OpenStack. It is used for deploying and managing large numbers of virtual machines and other instances to handle computing tasks.
- **Swift** is a storage system for objects and files.
- **Cinder** is a block storage component.
- **Neutron** provides the networking capability for OpenStack. It helps to ensure that each of the components of an OpenStack deployment can communicate with one another quickly and efficiently.
- **Horizon** is the dashboard behind OpenStack.
- **Keystone** provides identity services for OpenStack.
- **Glance** provides image services to OpenStack. In this case, "images" refers to images (or virtual copies) of hard disks. Glance allows these images to be used as templates when deploying new virtual machine instances.
- **Ceilometer** provides telemetry services, which allow the cloud to provide billing services to individual users of the cloud.
- **Heat** is the orchestration component of OpenStack, which allows developers to store the requirements of a cloud application in a file that defines what resources are necessary for that application.

5 The HW Architecture Layer

5.1 HPC Computing Resources

5.1.1 BSC Computing Resources

The Barcelona Supercomputing Center provides a set of high-performance computing resources to study how the most relevant performance limitations of biomedical applications can be effectively removed on modern HPC infrastructures. By using these cutting-edge high-performance systems, the DeepHealth project will demonstrate how its software toolkit is able to efficiently exploit heterogeneous HPC infrastructures. BSC hosts several HPC machines, being Marenostrum 4 the most relevant one. Marenostrum 4 has two separate parts: a general-purpose block and a block featuring emerging technologies. The emerging technologies block is formed of 2 clusters with different technologies and has the aim of providing computational services with the most advanced pre-exascale computing devices. Moreover, BSC provides another HPC cluster called Dibona, which is not part of Marenostrum 4. Altogether, BSC provides 4 HPC systems with different characteristics:

1. The general-purpose block of Marenostrum 4 consists of 48 racks with 3456 nodes of two Intel Xeon Platinum chips, each with 24 processors running at 2.1 GHz. The whole cluster sums up a total of 165,888 processors and 390 Terabytes of main memory, and is capable of reaching a peak performance of 11.15 PetaFLOP/s. The nodes are interconnected by a low-latency Omnipath network with a fully connected fat-tree topology.
2. One of the blocks of emerging technologies present in Marenostrum 4 combines IBM POWER9 CPUs and NVIDIA Volta GPUs. This cluster is composed of 54 nodes, where each node is equipped with 2 POWER9 processors, 4 Volta GPUs and 6.4TB of NVMe. The nodes are like the ones in the Sierra supercomputer at Lawrence Livermore National Laboratories, which is the 3rd fastest supercomputer in the top500 list. This cluster is very suitable both for HPC and for machine learning workloads, as it reaches a peak performance 1.57 PetaFLOP/s in double precision computations.
3. The second block of emerging technologies of Marenostrum 4 will be deployed in the next year, and it will be composed of Fujitsu A64FX 64-bit ARMv8 processors. The whole cluster will have a total computational capacity of over 0.5 PetaFLOP/s, and its nodes will have the same architecture as the future post-K supercomputer in Japan. The Fujitsu A64FX processor that will form the nodes of the cluster consists of 48 cores with a process technology of 7nm and 4 stacks of 8GB HBM2 memory, for a total of 32GB per node. This processor targets many different types of Exascale workloads, delivering a peak performance of 2.7 TeraFLOP/s of double precision compute power, 5.4 TeraFLOP/s in single precision, 10.8 TeraFLOP/s in half-precision, and 21.6 TeraOP/s in 8-bit integer precision. The architecture also includes new 512-bit SVE extensions with specific instructions for machine learning, making it a very appealing cutting-edge system for the workloads used in the DeepHealth project.
4. Dibona is a prototype HPC cluster designed by Arm, BSC and Bull (Atos Group) in the context of the Mont-Blanc project. Following the Mont-Blanc philosophy, Dibona is an Arm-based system composed by Cavium ThunderX2 processors. The cluster has 48 nodes with 2 ThunderX2 CPUs each, and can reach a theoretical peak performance of 49 TeraFLOP/s. The ThunderX2 processor contains 32 ARMv8 cores running at 2GHz with 128-bit NEON SIMD extension, a 32MB last-level cache, and different integrated hardware accelerators for security, storage, networking and virtualization. In terms of memory capacity, each node includes 128GB of DDR4 main memory with 8 channels per ThunderX processor and a 128GB SSD disk for local storage.

5.1.2 UNITO Computing Resources

UNITO will provide the access to the Occam cluster operated by the Competence Centre on Scientific Computing of the University of Turin (see Figure 8). Occam is a 1200-core heterogenous cluster composed of 32 “light” nodes (dual Xeon E5 12-core, 128GB), 4 “fat” nodes (quad Xeon E7 12 –core, 768GB), and 4 “GPU” nodes (dual Xeon E5 12-core, 128GB, 2 Nvidia P100). The key peculiarity of the Occam cluster is its management system, which has been designed and developed at UNITO [12]. Specifically, the Occam management system makes it possible to dynamically create bare-metal virtual clusters (called virtual farms), which are booked with a calendar-based system and deployed with a user-defined docker image. This makes it possible to run parallel application according to different deployment models, including SLUM and PBS queues, direct co-allocation of a set of nodes, etc. More information can be found at C3S web portal [13]. Occam is equipped with a 300TB Lustre “scratch” storage and a 1PB “archive” storage.

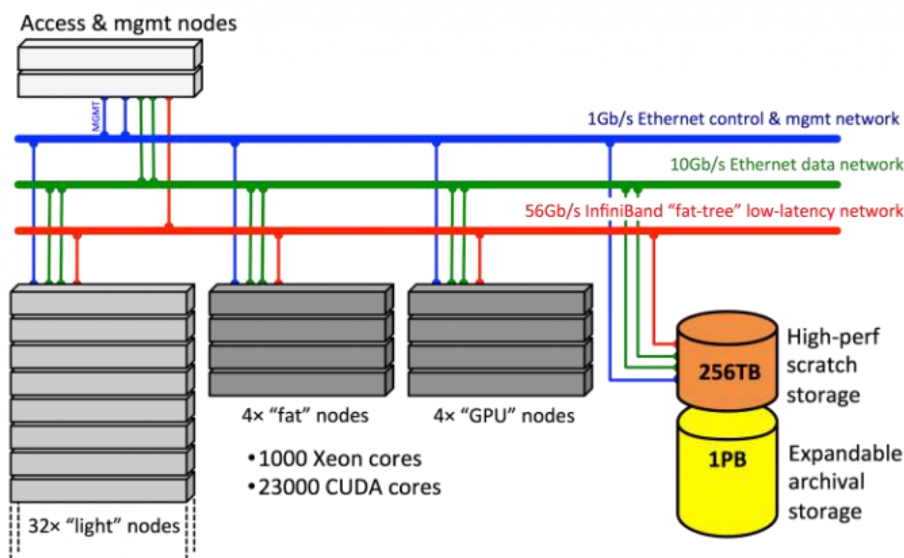

Figure 8. The Occam platform at UNITO.

5.1.3 UPV Computing Resources

UPV premises have a Xilinx ALVEO U200 PCIe card (see Figure 9). This FPGA is one of the most advances Xilinx boards available in the market and will be used to port and test the EDDL and ECVL libraries with the Xilinx workflow. The ALVEO U200 card has on-chip memory of 35MB and an off-chip memory of 64B. The ALVEO implements PCIe 3rd generation with 16 lanes to provide a bandwidth of 77GB/s and can perform 18.6 Tera INT8 operations per second.

Figure 9. The Xilinx Alveo Board.

To increase the performance that can be achieved with the latest Xilinx devices, UPV also plans to adapt the EDDL library to work in FPGA-based cloud environments. To that end, UPV will use one of the available cloud services that offer FPGA computation services using ALVEO cards (e.g AWS F1

or nimbix) to show the scalability to the EDDL library with FPGA acceleration using the most advanced technology.

As commented previously, the MANGO prototype will be used in the DeepHealth project. The MANGO prototype available at UPV premises has the following configuration:

- 8 supermicro servers
- 1 Gigabit switch
- 8 clusters of FPGAs, each with
    - 12 FPGAs
    - 8 DDR memories
    - 1 PCIe connection

Each cluster is built using a prototype solution from partner PRODESIGN, enabling the connection of 12 FPGAs and 8 DDR memories. Figure 10 shows a cluster.

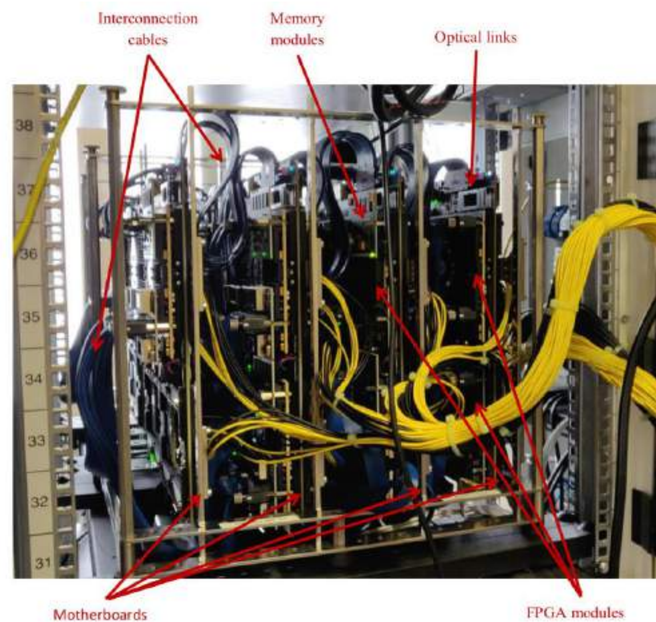

Figure 10. MANGO cluster photo.

The prototype has 78 KINTEX FPGAs, 8 ZYNQ FPGAs, 8 VIRTEX FPGAs, 1 STRATIX10 FPGA, and one ULTRASCALE+ FPGA. In total, 96 FPGAs and 64 DDR memories (see Figure 11).

Figure 11. MANGO prototype photo.

In addition to the previous infrastructure, UPV will also consider porting and using more advanced solutions based on FPGAs. One exploratory path is the use of the Cloud FPGA solution provided by IBM. This solution guarantees full throughput between FPGAs and provides a basic MPI-based communication infrastructure. This exploratory path will be analysed by UPV.

5.1.4 PRODESIGN Computing Resources

PRODESIGN will provide a FPGA based PCIe board. The FPGA will be a XILINX Ultrascale+ FPGA with embedded HBM memory (one of VU3P). The board furthermore will contain DDR4 SDRAM memory modules for additional DRAM extensions.

For host communication it is planned to deliver three types of interfaces:

- USB interface for board management and status observation
- MMI64 communication interface for general purpose application specific communication
- Data streaming interface for DMA based data exchange with the application providing the highest throughput

With the boards comes a management tool which allows to program the FPGA bitstream and to perform some status monitoring. The tool communicates with the board via USB.

For application specific communication Linux device drivers and C APIs will be provided for the MMI64 and the data streaming interface.

5.2 Cloud-based Computing Resources

5.2.1 TREE Computing Resources

In several situations, hybrid cloud solutions are a suitable solution to solve most common use cases in SME organizations. TREE hybrid cloud is an IaaS solution, that is a way to have an elastic hardware infrastructure easily adaptable to different workloads. TREE hybrid cloud solution will consist of:

- A private cloud which is composed of a set of on-premise hosts: it will be composed of a cluster of 5 nodes:
    - 4 nodes of x86 hosts (all of them with 4 cores and one of them with 16GB RAM and the other three 12GB RAM)
    - 1 x86 host (10 cores and 64 GB RAM) with a GPU (Nvidia Pascal Titan X, 12Gb GDDR5)
- Several public cloud providers which allow to make grow up the private cloud when it is necessary. There are a big variety of possible types of hardware which could be used from public cloud providers (I.e. Hosts of different sizes or GPUs). AWS, Azure and Google Cloud provide IaaS solutions.

5.2.2 UNITO Computing Resources

The HPC4AI cloud[3] at UNITO operates a federated version of the OpenStack cloud. Specifically, HPC4AI implements a zone of the GARR cloud the Italian national consortium of research. HPC4AI is currently composed of 10 nodes (Xeon 40-core, 512GB RAM, 4 NVidia T4 GPUs) but is planned to grow up to 40 nodes of the same kind mid 2020. HPC4AI offers the standard IaaS services of the OpenStack cloud and exploit a novel Deployment-as-a-Service service based on the Juju software (from Canonical).

[3] https://hpc4ai.it

6 Conclusions

The computing infrastructure upon which the DeepHealth EDDL and the ECVL libraries will execute have been described in this document. Concretely, the document incorporates: (1) the programming models (and access methods) for an efficient exploitation of the performance capabilities of the DeepHealth computing infrastructure, and (2) the HPC and big-data cloud based computing resources available for the project. This set of software and hardware components described in this deliverable will be the baseline upon which WP5 activities will be developed.

Overall, tasks 1.3 and 1.8 has been carried out successfully and the related project objectives have been reached and documented in this deliverable.

7 Bibliography

[1] Khronos Group, “https://www.khronos.org/opencl/,” 2019.

[2] NVIDIA, “https://developer.nvidia.com/cuda-zone,” 2019.

[3] OpenACC, “https://www.openacc.org,” 2019.

[4] OpenMP ARB, “https://www.openmp.org,” 2019.

[5] BSC, “https://pm.bsc.es/ompss,” 2019.

[6] Apache Hadoop, “https://hadoop.apache.org,” 2019.

[7] Apache Spark, “https://spark.apache.org,” 2019.

[8] BSC, “https://www.bsc.es/research-and-development/software-and-apps/software-list/comp-superscalar/,” 2019.

[9] Apache jcloud, “https://jclouds.apache.org,” 2019.

[10] Docker Inc., “https://www.docker.com,” 2019.

[11] Apache Mesos, “https://mesos.apache.org,” 2019.

[12] Open MPI, “https://www.open-mpi.org,” 2019.

[13] openstack, “https://www.openstack.org,” 2019.

[14] M. Aldinucci, S. Bagnasco, S. Lusso, P. Pasteris and S. Rabellino, “OCCAM: a flexible, multi-purpose and extendable HPC cluster,” *Journal of Physics: Conf. Series (CHEP 2016),* 2017.

[15] “C3S@UNITO web,” [Online]. Available: https://c3s.unito.it/.