DEEPHEALTH

D5.1 Efficient HPC Infrastructure for DeepHealth Libraries

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:	WP5/T5.1, T5.2, T5.3
Dissemination level:	PUBLIC
Document due Date:	01/04/2020 (M15)
Actual date of delivery	22/04/2020 (M16)
Leader of this deliverable	UPV
Author (s)	UPV, BSC, UNITO, PROD, EPFL
Version	v1.7

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
1	18/03/2020	First draft contents
1.1	02/04/2020	All contributions collected and integrated
1.2	03/04/2020	Revised by UPV and submitted for official review in the project
1.3	11/04/2020	Revised by UNITO (official review) and most suggestions added
1.4	18/04/2020	Final version with some small fixed suggested by UNITO (official review)
1.5	20/04/2020	Revised by Technical Manager
1.6	20/04/2020	Revised by Project Coordinator
1.7	22/04/2020	Final version to be submitted

Table of contents

1 Executive summary

In a sense, the goal of WP5 is to develop all the adaptations needed for the efficient use of HPC infrastructures in the DeepHealth project. This means adapting not only current supercomputers such as Marenostrum, but also technologies such as GPUs to a lower extent and FPGA to a much greater extent. This is seen in the project as providing heterogeneity support to the DeepHealth project. Also, all runtime adaptations need to be performed in this WP. This deliverable focuses on three specific tasks involving:

- Developing heterogeneous support (T5.1)
- Communication optimizations (T5.2)
- Synchronization optimizations (T5.3)

These tasks started at different project months, and because of that some tasks have been active for a longer period. This is the case for T5.1 focusing on the initial heterogeneous support developments needed for the adaptations of the DeepHealth project libraries that will be used in the target platforms. These are both the EDDL and ECVL libraries. The adaptation of those libraries is being carried out in WP2 and WP3, respectively for EDDL and ECVL. In this WP we deal mainly with the hardware and low-level software adaptation, a design space exploration of different configuration alternatives and the decision strategies to be taken at infrastructure level.

The deliverable goes smoothly on different initially unrelated topics as they focus on different specific activites by partners but mainly because they address either different hardware platforms or they address different software/runtime layers. First, we will describe in this deliverable all activities being carried out in order to provide support for the MANGO prototype which is made of 96 FPGA boards for the project. A general introduction to the prototype together with developed activities for its adaptation and customization to the DeepHealth support are described. After this, we describe the current activities within the project in order to provide partners with a specific FPGA board customized for the project. GPUs support activities are also briefly described next. Notice the adaptation of EDDL libraries is being performed by UPV (PRHLT group) and will be reported within WP2.

In a different direction, we devote two sections for the description of the activities evolved towards support for resource management of HPC infrastructures and how this links with other deliverables within the project, specially D5.4.

Finally, in a third dimension of the deliverable, we describe the activities towards the exploration of performance and energy impact of the current EDDL/ECVL libraries on CPUs, GPUs and FPGAs. For the case of CPUs and GPUs we introduce a detailed report of initial tests performed by BSC partner on their systems on premises when running a real use case with current EDDL library available within the project. For the FPGAs we delve into the different platforms and implementation options available for the correct development of compute intensive and energy efficient implementation of kernels for both EDDL and ECVL libraries. This work is being performed by UPV.

It has to be mentioned all these activities are related mainly to T5.1 in a great extent and in minor one to T5.2 and T5.3. It is expected that the next version of this deliverable will grow in the direction of communication and synchronization activities from T5.2 and T5.3 which have been active only for a few months. Indeed, this report will be updated accordingly in the third year of the project. For each activity reported we provide a subsection with planned activities for the next months in the project.

As a final comment, this deliverable has not been impacted severely by the world-wide situation occuring due to the COVID-19 pandemic. However, current activities, mostly in Italy and Spain, have been affected and potentially may impact on the expected delivery of the activities in this WP in general and next versions of this deliverable and associated tasks in particular. Next, we describe the situation of some key partners involved in associated tasks of this deliverable:

- As of today PROD did some preparations to minimize the risk and impact of the Corona virus to the company and the DeepHealth project. For example, most of the engineers are working remotely. Remotely controllable and remote access to the hardware in the labs is established. Also the production is working using a split shift approach to avoid a complete company shutdown in case of infected people. Except for the board production and initial setup and test, the majority of the current and outstanding work can be (and currently will be) done remotely. This means that as long as the people assigned to DeepHealth are not infected no impact to the project is expected. This leaves the production, including the potentially limited availability of material, as a main risk.
- As of today, UPV is impacted by the COVID-19 and every team member is working (the one associated with FPGAs) from home and in close collaboration with others via remote teleconferences. However, access to FPGA devices and to the MANGO cluster has not been allowed for the last month and is uncertain when activities will resume. There is a strong need of physical access to the FPGA devices for the correct operation and advancement. It has to be seen yet which will be the impact on the UPV work assigned through WP5.
- As of today, UNITO is following the rules dictated by the national and local authorities to provide the adequate preventive measures for the community to limit the spreading of the COVID-19. This includes the requirement to stay at home, limiting movements only for proven work needs. Therefore, team members are working remotely and remote access to the hardware infrastructure is guaranteed. Hardware updates for HPC4AI infrastructure that were scheduled to be completed by the end of March, will have at least a three months delay. However this has not affected the project activity so far.

2 Heterogeneous Support

In this Section we describe all the adaptations performed to use heterogeneous architectures for the DeepHealth project libraries and applications. First, we focus on the MANGO prototype which is positioned as one key technology architecture for the DeepHealth project. The work around this prototype will let assess the potential benefit of multi-FPGA clustered devices on the EDDL and ECVL libraries. The MANGO prototype part selected for its usage in DeepHealth is described first. Then, we focus on the communication infrastructure being developed to let the MANGO prototype be efficiently used in DeepHealth, by adopting OpenCL programming model and preparing the software for its use in the prototype. Notice that the runtime being deployed in this WP for MANGO will be used by the EDDL/ECVL libraries developed in WP2 and WP3, respectively. Then, we describe the physical adaptations of the prototype (received at UPV premises in fall 2019) in order to be ready for normal usage in DeepHealth project. Then, we describe the on-going activities for the achievement of a DeepHealth own FPGA boards specifically tailored for the project.

2.1 The MANGO Prototype

The goal of the MANGO project [1] consisted in developing, managing and exploring different promising architectures for future HPC systems. To achieve this purpose, the MANGO consortium proposed an **emulation platform** composed of General-purpose Nodes (GNs); clusters of FPGAs, as well as Heterogeneous Nodes (HNs); a Gigabit Ethernet interconnect among GNs and an innovative cooling and power monitor subsystem. This prototype is illustrated in Figure 1. In the MANGO project, those HNs offer the power to **instantiate different hardware architectures coded directly in HDL programming languages**. For managing both the different computational units implemented on a given architecture and the architecture itself, the consortium members developed **a low-level runtime library and extended a runtime management system**, which runs on the GN. It lets HPC applications offload kernels to the units implemented in a specific architecture, with the aim of **analyzing the impact of the implemented architecture for the executed application in the so called 3P space– performance, power and predictability –** which it is very valuable for the HPC community.

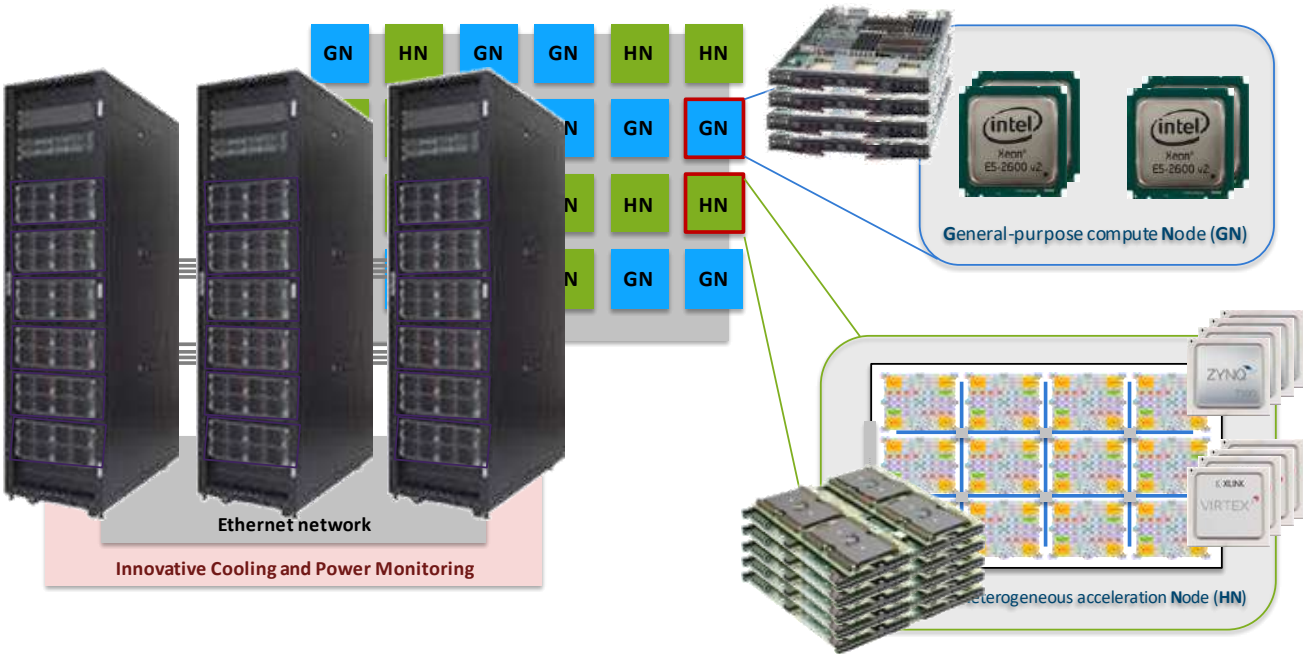

Figure 1 The MANGO prototype with its hardware components: I) General Purpose Nodes (GNs), II) cluster of FPGAs, as known as Heterogeneous Nodes (HNs), III) Gigabit Ethernet interconnect, IV) innovative cooling and power monitoring subsystem.

Among the different components integrated in the MANGO prototype, we focus on describing the GN and HN in this section, since they act as main enablers to provide the performance power required for accelerating the EDDL and ECVL libraries in Deep*Health*.

The GN consists of a blade made of a high-end CPU and a GPU. The specification of SBI-7128RG-F/F2 [2] can be found in Table 1. This blade uses a Xeon E5 V3 generation processor and can connect two PCIe 3.0 x16 (standard) or four PCIe 3.0 x8 boards (with a riser card). The F and F2 variants have a single or dual FDR port respectively. The blade is also equipped with 64 GB of DDR4 memory and 1 TB SSD hard disk.

Table 1 Server blade specification of the MANGO GN

Model	SBI-7128RG-F
Processors	Intel® Xeon® processor E5-2600 v3 to 9.6 GT/s(Haswell) product family with QPI up
Chipset	Intel® C610
Memory Support	LRDIMM / RDIMM DDR4-2133MHz in 8 slots
Max Memory	Up to 512GB LRDIMM, 256GB ECC RDIMM
Expansion & Hard Disk Drive	• Two Xeon Phi or Tesla Kepler K10X/K20M/K40M/K20X/K80, GRID K1/K2 • 4 PCI-E 3.0 x8 cards, • 2x SATA DOM or 1x SSD; or, • up to 8x 2.5 SATA3 HDDs + 1x 2.5" SSD**
InfiniBand/10GbE option	• X: Onboard 2-port 10 GbE • F: Onboard 1-port FDR InfiniBand* • F2: Onboard 2-port FDR InfiniBand*
Ethernet Interface	Intel® i350 dual-port Gigabit Ethernet Controller
Management	IPMI 2.0, KVM over IP, Virtual Media over LAN
Graphics	Aspeed AST 2400 VGA
Dimensions	288 x 42.5 x 520.7mm

The HN consists of a total of 12 FPGAs and 22GB of DDR-3 and DDR-4 memory mounted on top of 4 proFPGA motherboards [3] from ProDesign GmbH. This cluster is also heterogeneous, since it is composed of different types of FPGAs: Xilinx Kintex Ultrascale (KU115) [4], Xilinx Virtex 7 Series (V2000T) [5], Xilinx Zynq 7000 SoC (Z100) [6] and Intel Stratix 10 (SG280) [7]. Every cluster is connected to a GN through PCIe Gen3 x8 lanes, thus every GN can access two different HNs, for a total of 24 FPGAs and 44 GB of DDR3/4 memory. Table 2 presents a summary of the technical characteristics of the different hardware components each HN is equipped with.

Table 2 Technical details of the hardware components of an HN

Name of Material	Technical Details
proFPGA quad Motherboard	Offering up to 32 extension sites Extension sites for up to 4 x proFPGA FPGA modules Up to 4400 signals for I/O and inter FPGA connection On board operating and host data exchange system Multi motherboard connectivity support
proFPGA Xilinx XC7V2000T	Using latest FPGA technology with the Virtex XC7V2000T Offering up to 12 M ASIC gates capacity Up to 1100 free user I/O for daughter board connection 8 extension sites with individually adjustable voltage regions 16 MGTs (12.5 Gbps) per FPGA Modules
proFPGA Xilinx KU115	Offers a maximum capacity of up to 7.9 M ASIC gates and 5520 DSP slices in a single FPGA. 6 extension sites, up to 585 user I/Os for daughter board connection 56 high speed serial transceivers (56 x GTH) running up to 16.375 Gbps. The extension sites offer individually and stepless adjustable voltage regions from 1.2V up to 1.8V
proFPGA Xilinx Zynq	Xilinx Zynq XC7Z100 FPGA Dual ARM® Cortex™-A9 MPCore™ with CoreSight™ 260 standard I/Os and 16 high speed serial transceivers USB 2.0, Gigabit Ethernet, ARM JTAG, DDR3 & SPI Flash Mix and match with other proFPGA FPGA modules
proFPGA Intel Stratix SG280H	Handling up to 20 M ASIC gates capacity Modular with 1 x Intel® Stratix® 10 SG280 FPGA 5760 floating point DSP blocks Up to 1026 free user I/O Up to 8 individually adjustable voltage regions Up to 1.0 Gbps single ended point to point speed
proFPGA Interconnection Cable	Length matched high-speed interconnectcable Connects any extension sites (all directions) Creates a bus between two extension site connectors Connects up to 147 I/Os (three I/O banks), 16 clock I/Os Maximum point to point performance 1 set = 4 cables (40 cm)
proFPGA DDR3 SDRAM Board 2 GB	up to 2GB DDR3 SDRAM Assembled with 4 x 2 or 4 Gb DDR-1600 modules 64 bit wide databus Data rate up to 1600 Mbps Length matched board design offering highest performance
proFPGA DDR4 SDRAM Board 2.5GB	Up to 2.5GB DDR4 SDRAM memory and can be accessed over a 80bit databus. Due to the length matched board design, the daughter boards can be used with performances of up to 2400 Mbps. Besides the board offers three push-buttons and 8 on board LEDs, which can we used for debugging purpose.
proFPGA QSFP Interface Board	2 x QSFP+ connectors Programmable clock generator for MGT_REFCLK pins Fixed clock generator (100 MHz) for MGT_REFCLK pins 1 x proFPGA top connector to make unused IOs available
PCIe interface	Complete PCIe gen3, 8-lane host-proFPGA interface kit PCIe gen3, 8-lane, host interface card proFPGA 8-lane, PCIe gen3 connector board

	PCIe gen3, 8-lane compatible cable, 3 meter length Data rate up to 8 Gbps per lane
proFPGA PCIe DMBI Kit	Complete PCIe gen3, 8-lane host-proFPGA interface kit PCIe gen3, 8-lane, host interface card proFPGA 8-lane, PCIe gen3 connector board PCIe gen3, 8-lane compatible cable, 3 meter length Data rate up to 8 Gbps per lane
proFPGA Motherboard Interconnect Cable (40 cm)	The proFPGA prototyping system supports combined operations of multiple motherboards. Each motherboard provides two connectors for this purpose. The cable provides the connections for MMI-64 communication and exchange of clock signals between motherboards. Additional motherboards may be added to the system by daisy-chaining of motherboard interconnect cables.

The whole MANGO prototype consisted of a total of 8 GNs and 16 HNs. However, the prototype was split in different parts at the end of the project and each part was delivered to a different partner of the MANGO consortium, with the goal of keeping the prototype operative for the service of science and innovation in Europe. UPV received half of the original prototype. Therefore, 4 GNs and 8 HNs, for a total of 96 FPGAs and 176 GB of device memory, will be available for Deep*Health*.

Although MANGO encompassed mainly the exploration of unconventional architectural solutions, basically evaluated at the functional level, the reconfigurable hardware MANGO subsystem – i.e., the HN – has been conceived to play a twofold role in the final HPC demonstrator. This situation is well captured by Figure 2, highlighting how *the same FPGA platform* can support both a physical **compute platform** and an **emulation platform.**

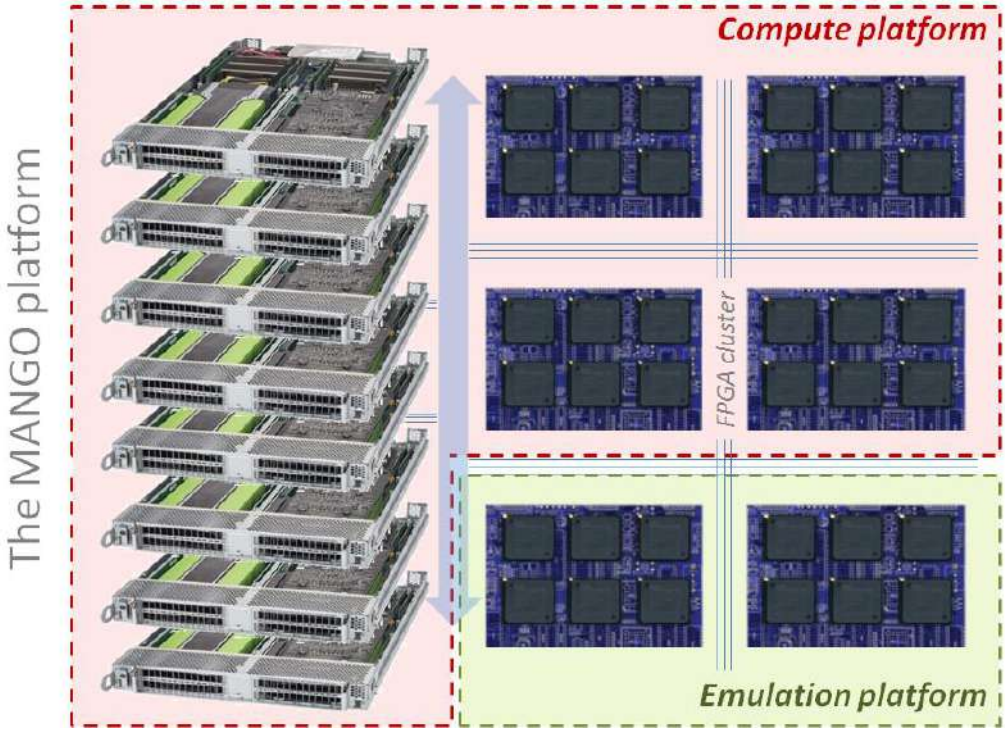

Figure 2 MANGO perspectives: I) Computation and II) emulation. As a computation platform the focus lays in bulk performance. As emulation platform the focus is on exploration and hardware solution prototyping.

In the tables below, we briefly summarize the key aspects covered by each of the two perspectives involving the role of FPGA HPC technologies:

FPGAs as a compute platform
Focus on bulk performance: special-purpose, accelerator-oriented approach
Tightly coupled to the system
Will be validated through dedicated accelerator designs
Will mainly expose non-functional characteristics, e.g. thermal and power behavior
Will allow direct validation of run-time management and cooling techniques
Note: The choice of the FPGA technology is critical (a balance between logic-dense versus high-performance must be a selection criteria)

FPGAs as an emulation platform
Focus on architectural exploration, both at the system level and compute unit level
Emulated systems will be general-purpose, software-programmable compute units and/or application-specific circuits.
Will mainly target functional aspects
Will allow the validation of the software stack
Physical performance will be inferred from suitable performance counters
Performance improvements will be measured in relative terms
Note: The choice of the FPGA technology is not critical (although logic-dense devices will be preferred)

Given the nature of the Deep*Health* project the FPGAs of the MANGO prototype will be used as a compute platform rather than as an emulation platform, with the aim of extracting the maximum performance of the libraries developed in the context of Deep*Health*.

2.2 MANGO Communication Support (runtime)

As compute platform providers, Xilinx, Intel and some of their related partners are currently the main vendors of FPGA cards targeting high-performance applications acceleration. They market different FPGA-based acceleration platforms with different specifications [8-9], but all those products are programmed following a similar design flow. The key goal of the flow lays in enabling a hardware platform and a set of software libraries to diminish programming complexities and facilitate the application of acceleration to software developers.

Usually, a program that runs on this type of system is composed of two pieces of source code, as illustrated in Figure 3. On the one hand, there is a piece of code that runs on the host and it is coded using a high-level programming language (e.g., C/C++) and compiled with a native host compiler (e.g., GCC). It contains the right sentences to initialize the device, transfer input data, offload kernels and get kernel data results back to the host. On the other hand, there exists another piece of code, as known as kernel, that it can be also coded in a high-level programming language like OpenCL, among others. Kernels are the part of the application that will run on accelerator devices so as to provide acceleration capabilities to the applications. This piece of code is compiled with a specific compiler for the target hardware platform. The Xilinx and Intel compilers for their hardware platforms are *xocc* and *aoc*, respectively.

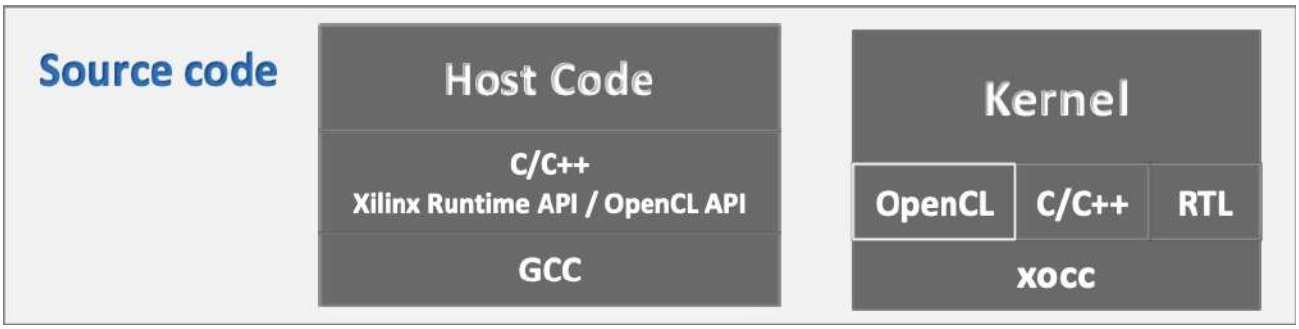

Figure 3 Structure of an application compatible to run in a Xilinx or Intel FPGA-based hardware platform. On the left, the code that will be run in the host, together with some of the programming languages supported. On the right, the kernel code that will be run in an accelerator device and some of the programming languages supported.

The design flow for a hardware platform mainly addresses two aspects. The first aspect aims to deploy a design inside of the FPGA consisting of two partitions, as it is shown in Figure 4. One of them is static, while the other is reconfigurable. The static partition is also known as *shell*. This partition provides the required elements to communicate with the host where the FPGA is connected in an efficient way. Usually, the communication is performed using a PCIe Gen3 x16 connector to get maximum communication performance, but any other server compatible communication interface could be used. As its name indicates, this partition is never changed while the host is up and running. In general, when the host is booting this partition is loaded into the FPGA from an internal flash memory located in the FPGA card. On contrary, the reconfigurable partition consists of a space inside of the FPGA that can be changed at runtime using the features that the *shell* provides. This partition contains the resources in which a hardware kernel can be deployed. In most recent Xilinx FPGAs, the DDR controller(s) is(are) mapped to this region as well. On the contrary, in Intel designs it is common that the DDR controller(s) is(are) included in the static partition instead. However, putting a DDR controller in the reconfigurable region makes the static partition smaller and facilitates the placement and routing of the reconfigurable partition allowing the implementation of more complex application kernels.

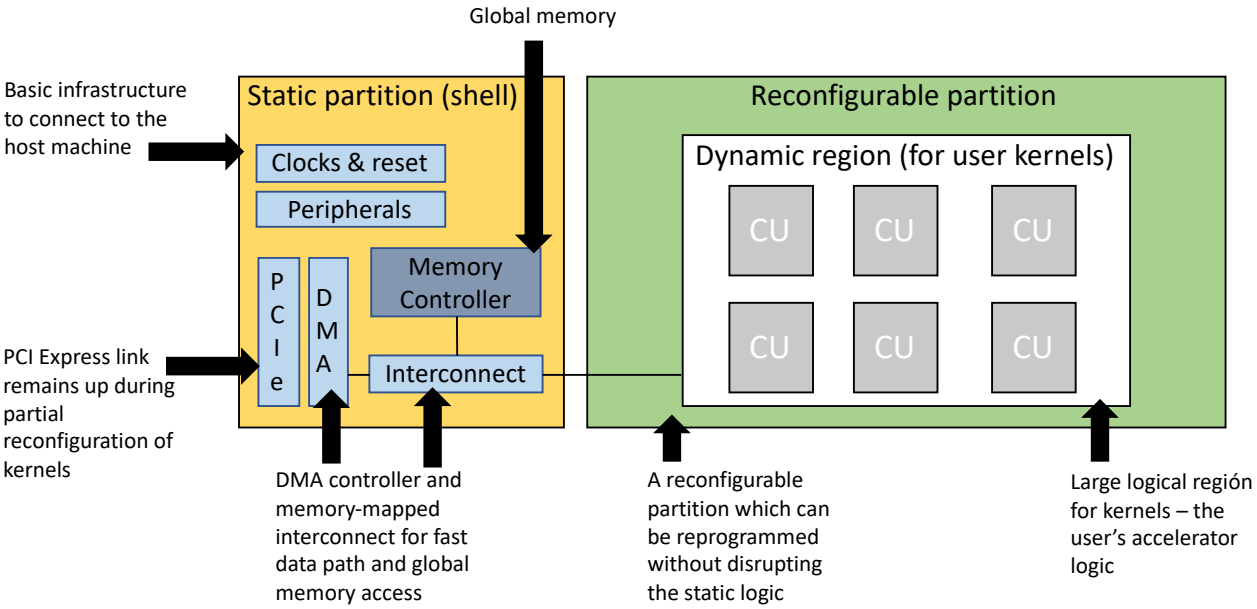

Figure 4 General block diagram of an FPGA design to support application acceleration. It contains: I) a static region with the common elements to communicate with the host for providing connectivity framework to access the programmable region and access memory, and II) the reconfigurable partition with enough FPGA resources to implement complex kernels that provide application acceleration.

The goal of the second aspect is to develop a complete software stack that includes from the required device drivers to user space libraries in order to reduce the programming complexity of these accelerator devices. As an example, Figure 5 shows such a stack for Xilinx devices. Each layer on the stack provides services to the upper layer with the aim of reducing programming complexity from layer to layer. From bottom to top we can see some Xilinx hardware platforms (blue boxes), the device drivers that run on kernel mode (orange boxes) and libraries and tools that run in user mode (red box), including a hardware abstraction layer (HAL) that hide programming complexity to applications, such that they can be still developed with traditional programming languages (pink boxes).

Both Xilinx and Intel offer the source code of different reference design models. These models include different block diagram designs, together with a portion of the software stack, which provides device drivers and a HAL compatible with them. These models can be used as a baseline to extend or adapt to other target requirements.

Contrary to the Intel and Xilinx approach, the FPGAs delivered within the MANGO prototype are naïve placeholders to implement the desired hardware or accelerator engines, but they do not come with any type of *shell* to provide a high-performance communication link with the host or facilitate the programming of these devices to software developers. Therefore, one of the activities that UPV is leading consists in **building a high-performance efficient *shell* to enable the MANGO prototype to accelerate the EDDL and ECVL libraries and facilitate the source code development task for acceleration to their software developers**. To do so, the goal is enabling OpenCL [10] and HLS [11] high-level programming models for the MANGO prototype

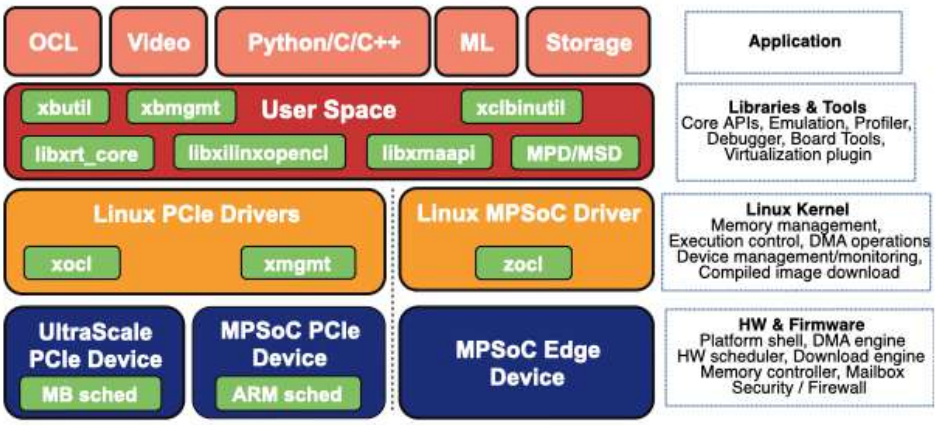

Figure 5 Xilinx software stack. A view from the high-level programming languages for user applications to the OS kernel drivers required to access hardware platforms.
Source: https://xilinx.github.io/XRT/master/html/index.html

Figure 6 shows the whole block diagram of the setup that we are currently implementing for the FPGAs that compose the HNs. Note that we are following an iterative process to deploy such a design, thus it could suffer slight modifications in forthcoming iterations with the aim of improving its efficiency. The design is split in two parts. One part contains a static logic (*shell*) that contains the essential modules to: enable bidirectional communication with the host through the PCIe IP core and with the other FPGAs of the cluster, generate clock and reset signals (shown with solid red lines in the figure) for the rest of the design subsystems and perform efficient data transfers between device and host memory, which is achieved implementing a direct memory access (DMA) module inside of the FPGA. The other part contains a reconfigurable partition which includes a DDR controller IP core to access 2GB of device memory and a placeholder for a set of application kernels (*User kernel* module) that can be offloaded from the host. In summary, we are following an approach similar to Xilinx and Intel to enable the HNs in the MANGO prototype as accelerator platforms. In fact, we are extending one of their reference models. Nevertheless, there are slight differences between the MANGO hardware platform and the Xilinx/Intel ones, as we comment next.

Although we have shown a design with the main elements, there exist other modules in the design that have not been shown. This is the case for a module that enables the debug of some elements of the system. This *debug* module has not been shown because it will be removed from the final version. Its current purpose is simply to provide a mechanism to debug and validate our design. In addition, currently there is only one FPGA that implements the PCIE IP core in each HN. The reason for that is that there is a single FPGA in each cluster that communicates with a GN through a PCIe adapter card. The other FPGAs in the cluster can be accessed through the *shell* of the single communication-enhanced FPGA by means of the *Smartconnect* module in the static logic part of the device. This is the main difference with the Xilinx/Intel designs. In those designs there is no need to provide a *shell* that connects different FPGAs among them, since every FPGA will be plugged in different PCIe slots of the host. On contrary, in our design there is only a primary FPGA in each HN that communicates with the host. Thus, the rest of FPGAs of each HN have to be accessed from the *shell* of the primary FPGA.

In order to implement our design, we are currently using the Xilinx Vivado 2017.1 [12] and the Intel Quartus 18.2 [13] tools. Both development environments require a license for their use. These tools offer similar functionalities and basically allow one to: generate a block diagram for a design, automate IP core connections, synthesize and implement a design, analyse timing closure and generate bitstream files for different FPGA family models.

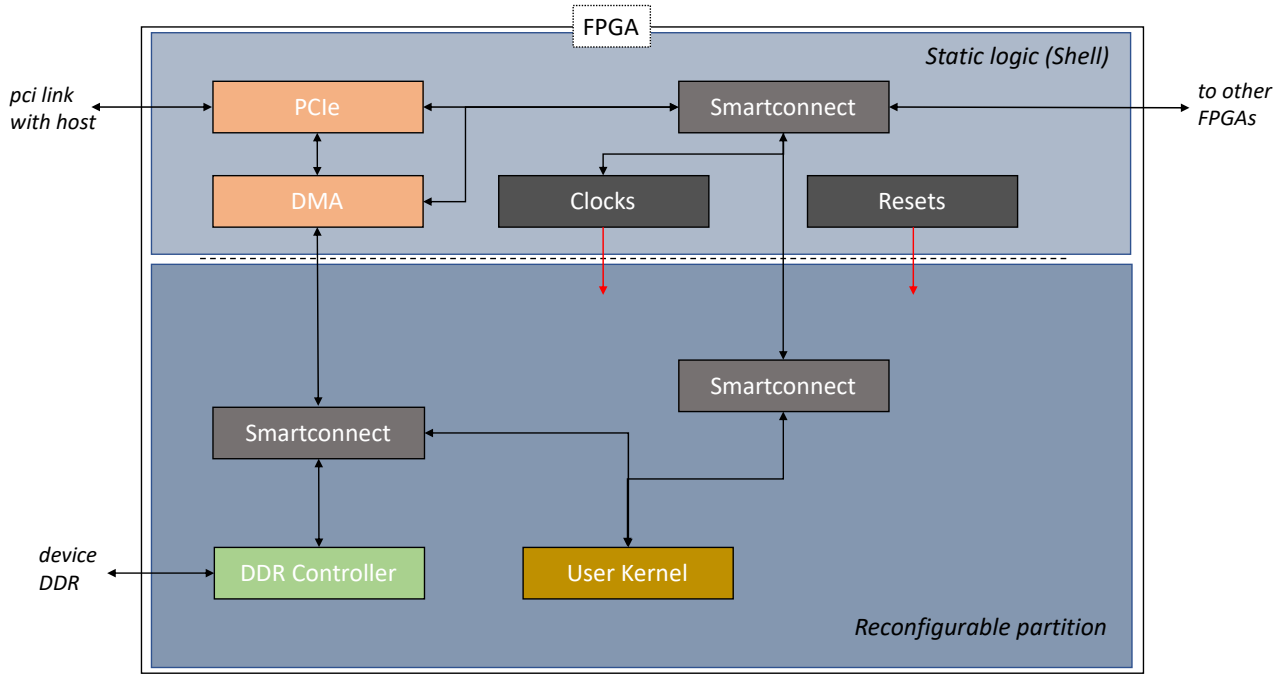

Figure 6 FPGA block diagram design overview. It contains the shell (static logic) and the reconfigurable partition, where the user kernel that provides acceleration for host applications will be placed by means of partial reconfiguration through PCIe.

With regards to timing closure, our design supports different clock frequencies. On the one hand, PCIe and DDR controllers work at a fixed frequency determined intrinsically by the underlying technology. The target frequency for both PCIe and DDR controllers is 250MHz. On the other hand, the kernel logic admits two clock frequencies for Xilinx designs and only one for Intel designs. The operative frequencies of kernels implemented for Xilinx devices are 250MHz and 500MHz, while the operative frequency is 333MHz for the Intel ones. A common feature of clock generators for both Xilinx and Intel is that they offer the possibility of performing frequency scaling to reduce power usage, when possible.

Figure 7 illustrates the resources occupied by the implemented design for a Kintex Ultrascale KU115 FPGA device. The highlighted portion of the device refers to the static region (*shell*). In this implementation, the static partition contains the PCIe core as well. Thus, this design is only compatible with configurations with a single FPGA in each HN – the one connected to the GN through PCIe. This figure clearly illustrates one of the most remarkable achievements of our shell implementation: it only occupies approximately 8% of the resources in that particular device.

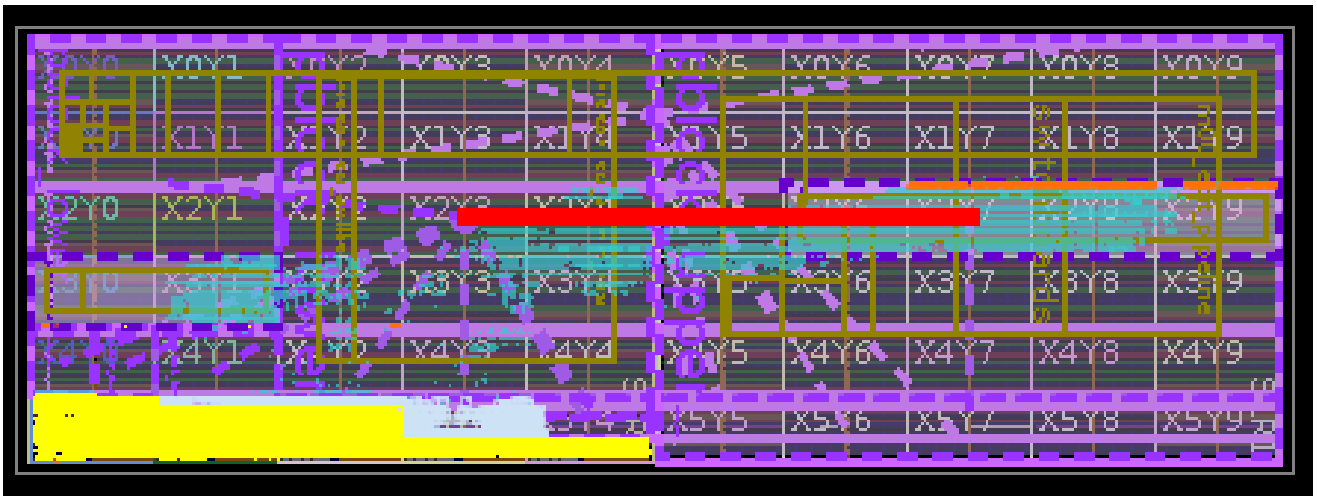

Figure 7. View of the implemented design for a Kintex Ultrascale 115 FPGA device. The highlighted portion represents the area occupied for the shell in that device. The figure has been rotated 90 degrees clockwise for improved readability of the document.

The communication with the FPGAs of every HN is carried out with the corresponding Xilinx and Intel PCI device drivers. As mentioned above, both Xilinx and Intel offer the source code of their HAL and drivers, so they can be adapted for working with implementations that are based on their reference models, as is the case of the one that we are developing for the Intel and Xilinx FPGAs of the MANGO prototype.

Basically, the driver presents a memory-mapped device to the upper layers. Thus, the access to the device is carried out by writing to memory locations. On top of the low-level device driver, a HAL Driver API is required by the used runtime, which is usually OpenCL, to communicate with the hardware platform. It is used for downloading FPGA bitstreams, allocating/de-allocating buffers, migrating buffers between host memory and hardware platform memory, and communicating with the kernel on its control port. The API supports address spaces, which may be used for accessing device peripherals with their own specific memory mapped ranges.

Figure 8 shows the current address space ranges for an FPGA of an HN. This address space is divided in two main groups. The first group (expanded_region/u_ocl_region) allows user kernels to access 2GB of DDR memory, starting from offset 0x0. The second group (base_region/dma_pcie) allows the host to communicate with different components in the design. This group is also divided in two categories: base_region/dma_pci/M_AXI and base_region/dma_pcie/M_AXI_LITE. On the one hand, we have proposed a 64-bit address space to allow the host to perform efficient memory data transfers to and from the device memory. This is achieved with the assistance of the DMA IP core. This core can access 2GB of DDR device memory, starting from the address offset 0. Thus, both the kernels and the host can access same memory address space. This is completely normal; since the host has to copy data first to the device memory and then, the kernel must perform its function with the copied data. On the other hand, we have proposed a 32-bit address space to access the other design components. In this sense, the host can access the kernel control port by using 128KB address range when writing at I/O memory address 0x0 and can access the kernel clock configuration ports through 4KB of allocated space for performing frequency scaling on the two available kernel clocks when writing at I/O memory address 0x0005_0000 and 0x0005_1000,

respectively. In the same way that the one described here, the rest of components of the design have their own memory range and offsets to access them in the memory space defined.

Cell	Slave Interface	Base Name	Offset Address	Range
expanded_region/u_ocl_region				
M_AXI (31 address bits : 2G)				
expanded_region/memc/ddrmem_0	C0_DDR4_S_AXI	C0_DDR4_ADDRESS_BLOCK	0x0000_0000	2G
base_region/dma_pcie				
M_AXI (64 address bits : 16E)				
expanded_region/apm_sys/xilmonitor_fifo0	S_AXI_FULL	Mem1	0x0000_0020_0000_0000	2G
expanded_region/memc/ddrmem_0	C0_DDR4_S_AXI	C0_DDR4_ADDRESS_BLOCK	0x0000_0000_0000_0000	2G
M_AXI_LITE (32 address bits : 4G)				
expanded_region/apm_sys/xilmonitor_apm	S_AXI	Reg	0x0010_0000	64K
expanded_region/apm_sys/xilmonitor_fifo0	S_AXI	Mem0	0x0011_0000	4K
expanded_region/u_ocl_region	S_AXI	Reg0	0x0000_0000	128K
base_region/pr_isolation_expanded/gate_pr	S_AXI	Reg	0x0003_0000	4K
base_region/featureid/gpio_featureid	S_AXI	Reg	0x0003_1000	4K
base_region/base_clocking/clkwiz_kernel	s_axi_lite	Reg	0x0005_0000	4K
base_region/pr_isolation_expanded/ddr_calib_status	S_AXI	Reg	0x0003_2000	4K
base_region/base_clocking/clkwiz_kernel2	s_axi_lite	Reg	0x0005_1000	4K
expanded_region/memc/ddrmem_0	C0_DDR4_S_AXI_CTRL	C0_REG	0x0006_0000	128K

Figure 8 Memory-mapped address space for accessing the different components implemented in every FPGA of an HN. The memory space is divided in two categories. The first category allows access to the kernels to 2GB of memory. The second category enables communication between the host and different components implemented in the FPGA, e.g. memory and kernel control port.

2.2.1 Planned activities

As commented above, we have proposed an iterative process in order to build the whole design of the FPGAs in every HN. In the first iteration we have already built the design for the FPGA connected to the GN through PCIe. In the next iterations, we will implement the design for the FPGAs that are not connected directly to any GN, and we will also extend the first design to add the logic required to communicate through the *shell* with the rest of the FPGAs that compose each HN, in a full duplex manner.

2.3 MANGO Prototype Physical Adaptation

The MANGO prototype targeting DeepHealth was received at UPV premises on the fall of 2019. Since then, we have been adapting and preparing the HW for the DeepHealth project. In this section we describe those adaptations.

With the aim to put the system back into operation, the hardware (GNs and HNs) had to be redistributed among two cabinets taking into account the location limitations imposed by the availability of only a single chassis for the servers and five power supply units (made up of five power supply units by muRata model D1U4CS-W-2200-12-HC4C and five power connector adapter cards by muRata model D1U4CS-12-CON) to power the eight HN systems.

The system came partially disassembled; only the HN systems were received assembled due to the complexity of disassembling the modules and the fragility of these components. Each HN system came properly packed in one box; the cabinets arrived packed on two pallets; all the remaining HW components (cables, cards, power supplies) came in separate pieces and packed in separate boxes. Figure 9 shows an HN system inside the transport box after removing the side and top protections.

Figure 9. HN system inside transport box.

To get the system up and running again various actions were needed. The first was performing a visual inspection of the received hardware to check the status of the received components. This was aimed at detecting any damaged hardware and to prevent any possible dangerous situation caused by powering-up damaged hardware. The second action was to design a power distribution network to provide power to the HN systems from the power supply units. This part of the hardware was not delivered to UPV premises since it is a custom prototype development, part of the contribution of EATON to the MANGO project, and it could be transferred due to safety issues. The third action was checking the hardware status by running tests that program the FPGAs and test cables to validate that all hardware is fully functional after the disassembly, transport and reassembly process.

2.3.1 Preliminary hardware check

The first step was to unpack all the material and perform a visual inspection to verify that all the components were in a good condition to be reassembled and powered up. In this stage we looked for any possible damaged and/or loose parts to prevent any potential risk for the personnel or the equipment. As a result, some damages were observed and fixed. Table 3 describes the detailed list of all checked elements. Following is a summary of the damaged components found in the inspection process and the actions taken/to-be-taken.

- CABINET – 1. Quick release mechanism of front door broken. This damage will remain as is, since it is not necessary to have the capability of removing the door.
- SERVERs – ALL. PCIe extension boards were loosening. All boards were reattached to the server frame. Since all blades were opened, all dust was removed from the inside, specially the heat sink (see Figure 10).
- HN SYSTEMs – Chassis structure. Loose screws. Screws were tightened.
- HN SYSTEMs – CABLES. Some interconnect cables were crushed and showed mechanical stress marks, and one had been peeled off. Connectivity of these cables will be checked.
- HN SYSTEM - 1 – Motherboard 2. The xusb1 connector, used for direct communications with this motherboard, was broken but still connected to the data paths. The connector must be reattached to the motherboard to avoid major damage to the device.
- CONNECTOR CARD of PSU unit. Power on/off switch was damaged and the contact pad did not hold its position firmly, leading to an unstable power output of the PSU as showed in Figure 10. The switch has been replaced by another one from the same manufacturer and series.

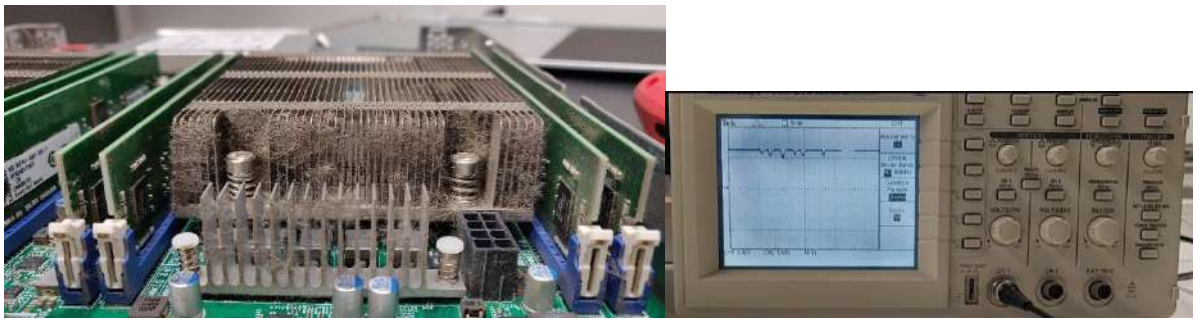

Figure 10. Dust in heat sinks in PU servers and PSU power output of connector card with damaged switch.

2.3.2 Hardware distribution among the cabinets

For the hardware distribution, the available cabinet setup was considered. CABINET – 1 includes two PDUs for mains power distribution, three trays, and three extra tray support guides. CABINET – 2 includes two PDUs for main power distribution, one chassis to allocate the GN servers, two trays and three extra support guides as well.

So, taking this into account, all the GN servers and two HN systems were placed in CABINET – 2, leaving space for the monitor-keyboard module and for the 12V power supply unit for the HN systems. The remaining six HN systems were placed into CABINET – 1, in groups of two HN systems per tray. The final hardware setup is shown Figure 11.

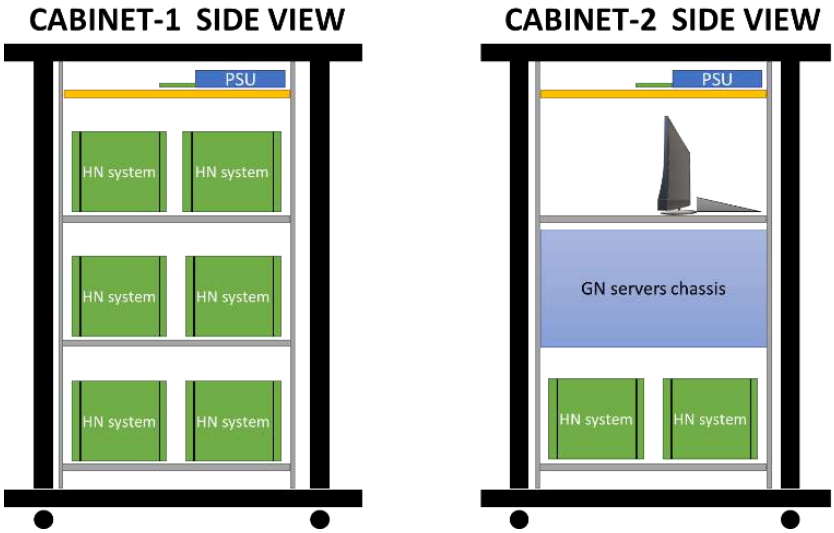

Figure 11. Hardware distribution in the cabinets.

2.3.3 12V power supply distribution network

To cope with the power supply distribution within the cabinets, a solution was designed and implemented taking into account: safety, power stability, ease of connection/disconnection of the HN systems from the power distribution network, power consumption of the current setup, and maximum power consumption of the HN systems in case of addition of new modules (up to the maximum number of devices supported by each HN system).

With current HN system setup, each HN system can demand up to 100 Amperes, while the current could reach 200 Amperes if the system were to be expanded with additional modules up to the maximum connection capacity of the motherboards; this load matches the maximum capacity of the power supply units. So, wiring and connectors were selected to support up to 200 Amperes. With these requirements, the section of the electrical cable to use was 70 mm2. The selected fast

connector for the HN system side was an APP 175; contacts are made for a 70 mm2 section cable and it is capable of supporting up to 340 Amperes @ 600Volts. The selected connection for the D1U4CS-12-CONC side was a terminal lung for a cable of 70 mm2. In order to avoid overlapping of the terminal lungs in the D1U4CS-12-CONC due to the position of the holes on both elements, as shown in Figure 12, connections with copper bars were added to maximize the contact surface (shown in the same Figure).

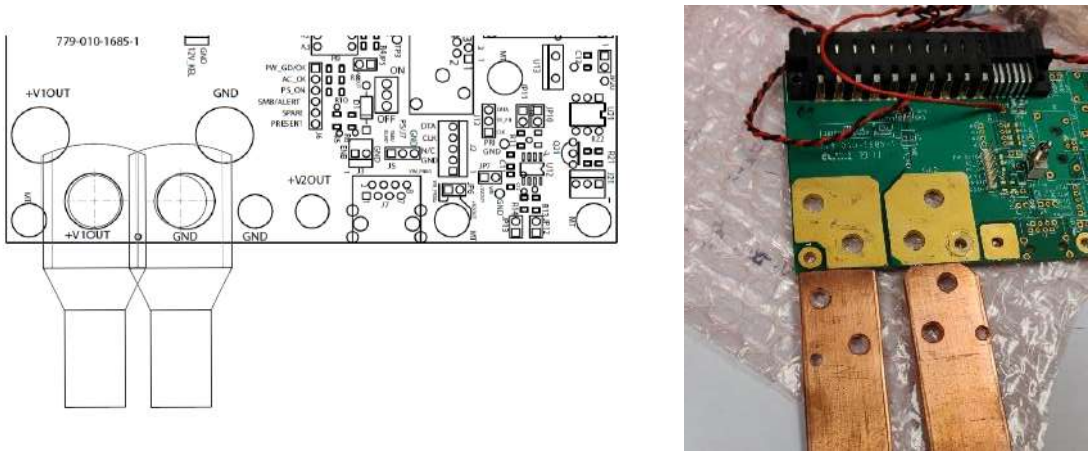

Figure 12. Terminal lung overlapping on D1U4CS-12-CONC.

D1U4CS-12-CONC board has power output pads on each layer: top and bottom. Therefore, two pairs of copper connection bars were attached to the board, one on the top layer and the other on the bottom layer; each layer will power one HN system. The selected cables for the power distribution network were RZ1-K 0,6/1KV (AS) 1X70 for the positive pole and ARAFLEX RV-K 0,6/1KV 1X70 for the negative pole. Then, the power will be distributed to all the FPGAs by means of a star-shaped power distribution network.

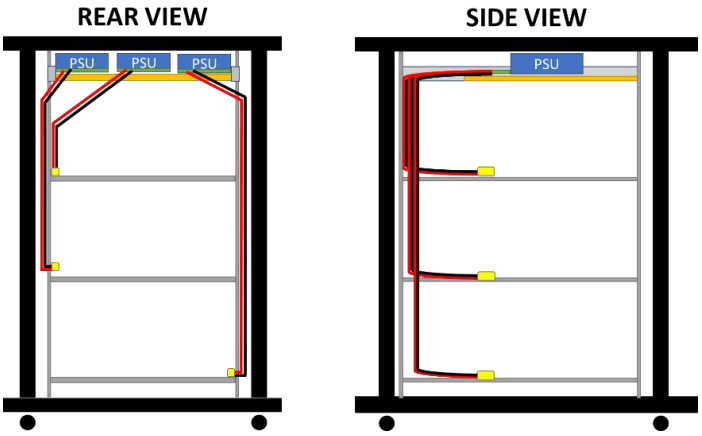

Figure 13. 12V power distribution network for CABINET-1.

Figure 14 shows the test setup, with one power supply unit providing power to two HN systems. Both HN systems are programmed with one of the setups used in MANGO project and have been running for 8 hours to test power system stability. This setup has been repeated for all the power supplies, connection cards, power wires and HN systems for full system check.

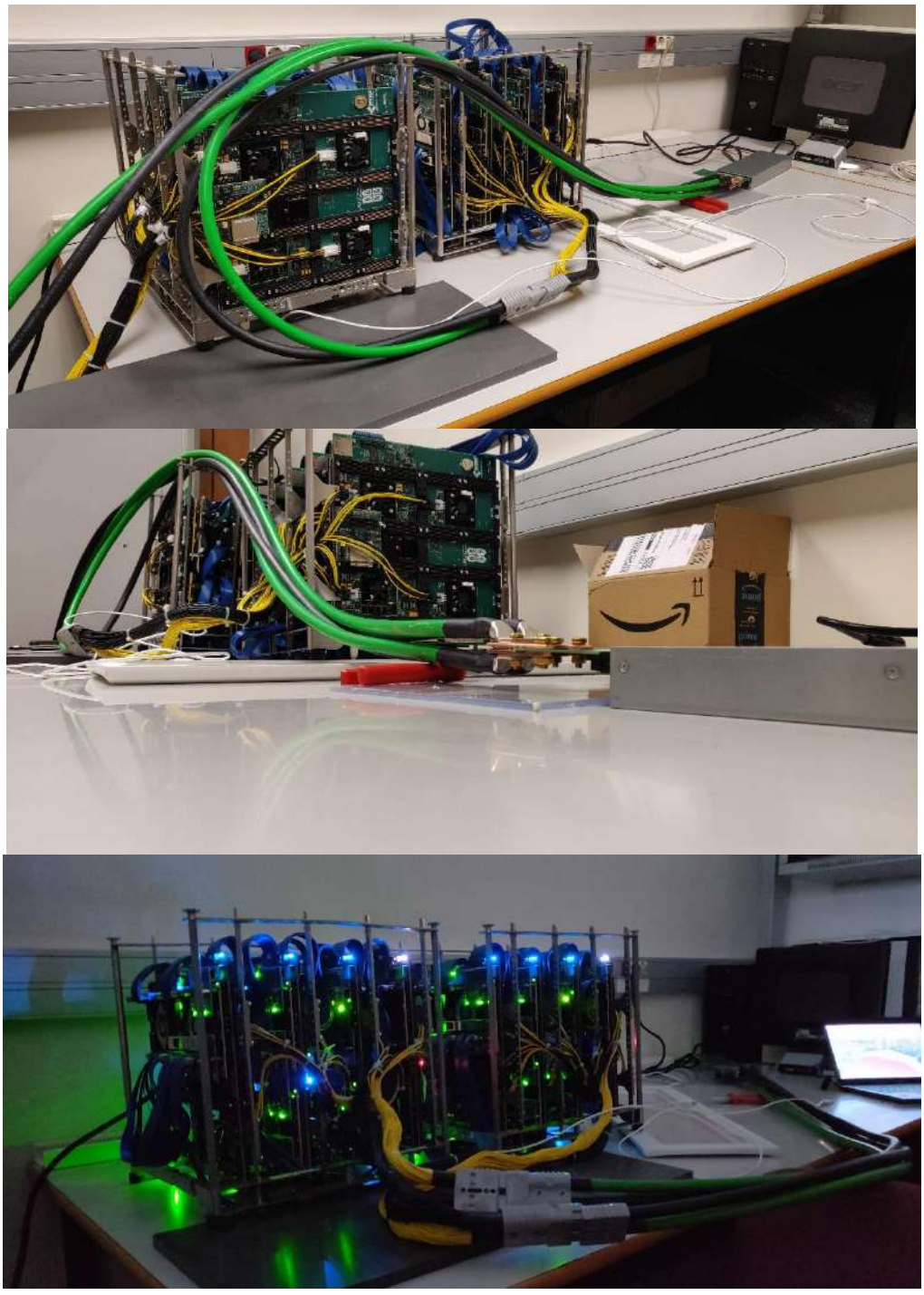

Figure 14. Two HN systems power-up setup.

2.3.4 Planned activities

Because of the-19 COVID emergency, the current activities for the physical adaptation and tuning of the prototype have been forcedly stopped. However, once a normal situation is restored the tests on the prototype will be completed and defective cables will be replaced by new ones. We expect this work to be completed within two months from when normal activity will be resumed. This delay may, however, impact the availability of the prototype for the project.

Table 3: Preliminary check of physical adaptations of MANGO prototype, received by UPV in fall 2019.

Component		Pass	Observations
CABINET - 1	chassis integrity and stability	yes	
	doors open-close-release mechanisms	NO	Front door release mechanism broken
	panels open-close-release mechanisms	yes	
CABINET - 2	chassis integrity and stability	yes	
	doors open-close-release mechanisms	yes	
	panels open-close-release mechanisms	yes	
SERVERS CHASSIS	Anchoring to the cabinet	yes	
	Connectors	yes	
SERVER - 1	Blade frame	yes	
	Connectors	yes	
	PCIe extension board anchoring	NO	Loos board, reattach board
			clean dust from heat dissipators
SERVER - 2	Blade frame	yes	
	Connectors	yes	
	PCIe extension board anchoring	NO	Loose board, reattach board
			clean dust from heat dissipators
SERVER - 3	Blade frame	yes	
	Connectors	yes	
	PCIe extension board anchoring	NO	Loose board, reattach board
			clean dust from heat dissipators
SERVER - 4	Blade frame	yes	
	Connectors	yes	
	PCIe extension board anchoring	NO	Loose board, reattach board
			clean dust from heat dissipators
PSU - 1	frame	yes	
	Connector	yes	
PSU - 2	Frame	yes	
	Connector	yes	
PSU - 3	Frame	yes	
	Connector	yes	
PSU - 4	frame	yes	
	Connector	yes	
PSU - 5	frame	yes	
	Connector	NO	Connector frame broken. Not an issue
CONN-CARD - 1	Board integrity	yes	
	ON/OFF switch	yes	
	connector	yes	
CONN-CARD - 2	Board integrity	yes	
	ON/OFF switch	yes	
	connector	yes	
CONN-CARD - 3	Board integrity	yes	
	ON/OFF switch	yes	
	connector	yes	
CONN-CARD - 4	Board integrity	yes	
	ON/OFF switch	yes	
	connector	yes	

CONN-CARD - 5	Board integrity	yes	
	ON/OFF switch	NO	Switch frame broken, unstable lever
	connector	yes	
PCIe cable - 1	Connectors	Yes	
	connectivity	-------	TO BE CHECKED
PCIe cable - 2	Connectors	Yes	
	connectivity	-------	TO BE CHECKED
PCIe cable - 3	Connectors	Yes	
	connectivity	-------	TO BE CHECKED
PCIe cable - 4	Connectors	Yes	
	connectivity	-------	TO BE CHECKED
USB HUB - 1	Frame	yes	
	connectivity	-------	TO BE CHECKED
USB HUB - 2	Frame	yes	
	connectivity	-------	TO BE CHECKED
USB HUB - 3	Frame	yes	
	connectivity	-------	TO BE CHECKED
USB HUB - 4	Frame	yes	
	connectivity	-------	TO BE CHECKED
USB cables			
HN - 1	Motherboards Connectors	no	MB2 xusb 1 connector teared off.
	Interconnect Cables	no	Some cables crushed
	Cluster integrity	no	Loose screws. Tightened
	FPGAs status	yes	proFPGA tool recognizes all components
HN - 2	Motherboards Connectors	no	
	Interconnect Cables	no	Some cables crushed
	Cluster integrity	no	Loose screws. Tightened
	FPGAs status	yes	proFPGA tool recognizes all components
HN - 3	Motherboards Connectors	no	
	Interconnect Cables	no	Some cables crushed
	Cluster integrity	no	Loose screws. Tightened
	FPGAs status	yes	proFPGA tool recognizes all components
HN - 4	Motherboards Connectors	no	
	Interconnect Cables	no	Some cables crushed
	Cluster integrity	no	Loose screws. Tightened
	FPGAs status	yes	proFPGA tool recognizes all components

2.4 New FPGA board

In this section we describe the ongoing activities for the development of a new FPGA board for the project. A high memory bandwidth of mid-size memories was identified as a key requirement for the new FPGA board. This results into the usage of an FPGA providing HBM memory, which is a FPGA internal DDR4 memory.

A second key requirement is the communication bandwidth between the FPGA and the host and between several FPGA boards. PCI Express was selected as the communication interface with the host. This interface is supported by almost all HPC servers.

Initially, PROD was in contact with INTEL to select the FPGA type which fulfils the requirements best. There were some investigations regarding upcoming FPGAs which are on the roadmap of INTEL. Based on the information gathered, it was decided to use the INTEL Stratix-10 MX1650 or MX2100 FPGA. The two models are package and pin compatible. The MX2100 provides 16GByte of HBM memory and the MX1650 provides 8GByte of HMB memory. The MX2100 is the preferred choice and will be used for the first boards.

PROD started a detailed specification of the board. The concept is shown in Figure 15. The board provides several connectors to extend the capabilities in a modular way.

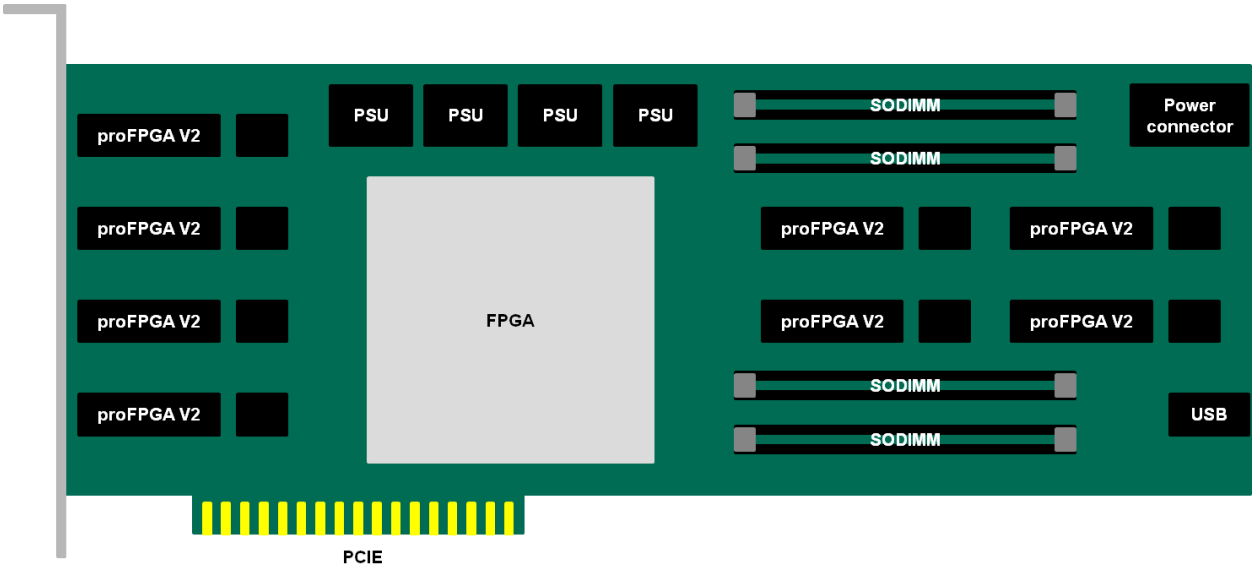

Figure 15: Concept of the new FPGA board

The SODIMM connectors are used for memories and peripherals which are connected to regular FPGA I/Os. In DeepHealth these SODIMM extension board sites can be used to attach additional memories to the FPGA depending on the applications requirements. Examples of such memories are DDR4 memory or high-speed SRAM memories to support random memory access with a minimal latency.

The Gigabit transceivers of the FPGA will be made available using the “proFPGA V2” connectors. These connectors allow data rates with more than 100 GBit/s per differential pair signal. Using these V2 connectors, interfaces like QSFP28 (see Figure 16) and Firefly can be attached, or connections to other FPGA boards using dedicated cables can be established. Those connectors allow scalability of the hardware in terms of capacity.

Since the PCI Express interface leads into the PCI form factor of the board, cooling is one of the major challenges. To reduce the risk of inadequate solutions some investigations were made to determine cooling options. A proFPGA FPGA board with a XILINX Virtex Ultrascale+ FPGA was used for some experiments. This board allows an adjustable power consumption of up to 200W.

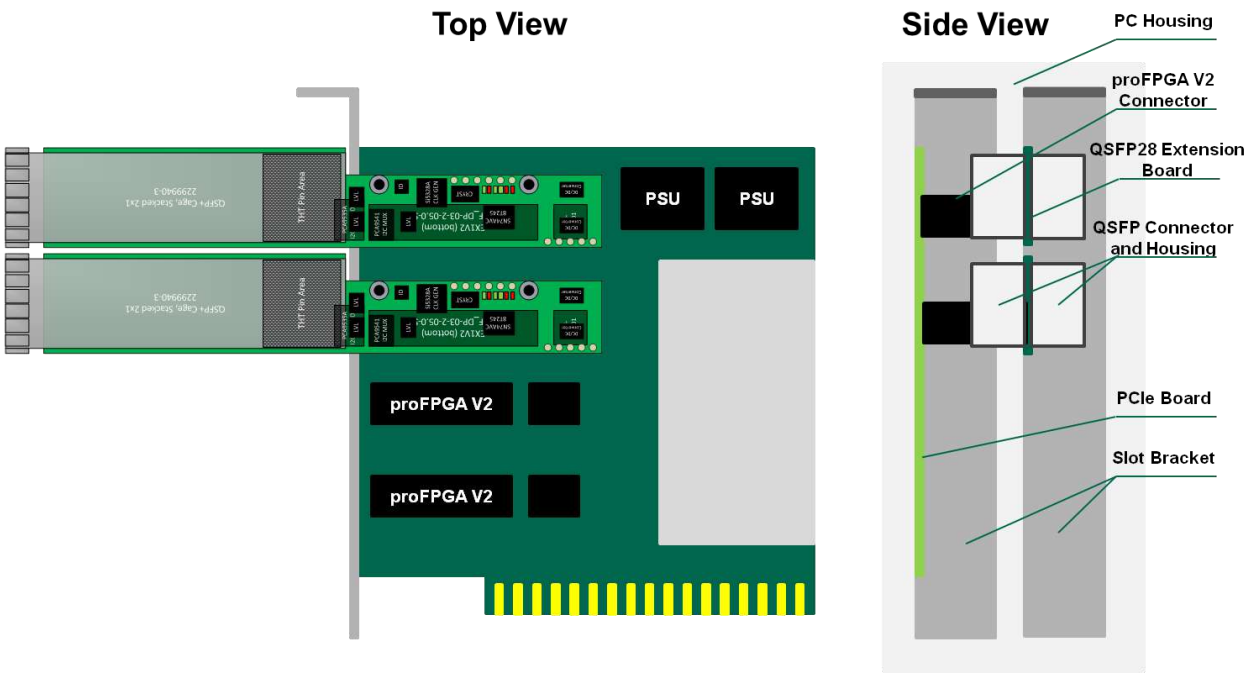

Figure 16: Case study - Extensibility with QSFP28 interfaces

Four different cooling options will be available. Some are interchangeable by the user. Some require a dedicated board assembly:

- Single slot passive cooling
- Single slot active cooling with fan
- Dual slot active cooling with fan (preferred cooling option for DeepHealth)
- Liquid cooling

Currently the detailed specification and the schematic design are in progress. As of today the availability of the first prototypes are estimated in Q4/2020. The biggest risks are the availability of the needed material and potential shutdowns due to the COVID-19 crisis.

In parallel with the board development, PROD is working on a PCI Express interface IP which is optimized for maximum throughput and lowest latency. Within DeepHealth there are different use models of this PCI Express interface depending on the FPGA hardware:

- MANGO hardware / XILINX FPGAs + PCIe extension kit
- MANGO hardware / INTEL FPGAs + PCIe extension kit
- DeepHealth PCIe board

The PCI Express interface IP consists of a PCIe controller, a Linux device driver, a software API and some service tools for configuration and status monitoring. The interface must support XILINX and INTEL in a way that the hardware API and software API are independent of the FPGA type.

First, a specification was created to describe all the components of the interface. The vendor-specific implementation details are also part of this specification. As of today, the implementation work is in progress. A first version for XILINX Ultrascale FPGAs is operational and will be optimized shortly. In a next step, the INTEL implementation will be added. They will be available when the first prototype of the PCI Express board can be provided to the partners.

DeepHealth partners also request to use the OpenCL/HLS tools provided by the FPGA vendor. It was confirmed by INTEL that these Vendor tools can be adapted to a custom hardware. Based on this feedback from Intel, it is assumed that OpenCL/HLS is supported by this PCI Express board. First tests with the MANGO hardware support this assumption.

2.4.1 Planned Activities

The prototypes of the boards, including the firmware (PCI Express interface, configuration, status monitoring) will be provided to the partners. While partners will be working on their implementation and tests, PROD will enter a phase of optimization.

It is expected that the throughput and latency of the PCI Express interface will require particular attention during optimization. Also, more extensive tests of the cooling options are planned.

These optimizations will be driven by feedback provided by the partners and by the test results made be PROD itself. A redesign of the board is planned too. The concrete date of the redesign depends on the progress and test results of the partners. Having enough applications already running on the prototypes would be very valuable.

2.5 GPU-based Support

The target architecture for the DeepHealth project provided by University of Turin (UNITO) is a hybrid HPC+cloud platform, also comprising GPU nodes and integrating DeepHealth libraries. Adaptations performed to enable DeepHealth applications to use the heterogeneous architecture are developed at the system level. They are made by configuring an HPC Secure Tenant (HST) and providing a workflow management tool, StreamFlow, to efficiently deploying and executing applications on different nodes according to their specific computational needs.

The HPC part of the platform is the OCCAM (Open Computing Cluster for Advanced data Manipulation) SuperComputer [16] cluster composed by:

- 42 heterogenous nodes, 1PB storage, 300TB High-Perf scratch
  - 2 management nodes: Intel dual-socket 24 cores, 64 GB RAM
  - 32 light nodes: Intel dual socket 24 cores, 128GB RAM
  - 4 fat nodes: Intel quad socket 48 cores, 768GB RAM
  - 4 GPU nodes: Intel dual socket 24 cores, 128GB RAM + 2 Nvidia K40
- Infiniband FDR, Ethernet 10G

The architecture schema is provided in Figure 17. OCCAM can be configured to provide an elastic virtual farm of containers, which are dynamically reconfigurable clusters using standard lightweight Docker containers.

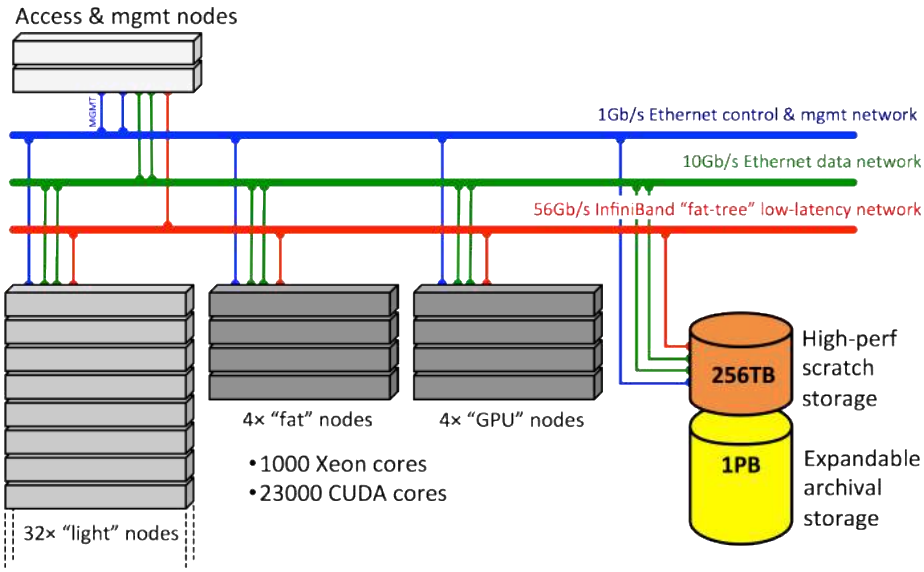

Figure 17. OCCAM Architecture Schema

The cloud part is the HPC4AI [17] infrastructure, which is a federated OpenStack [19] cloud with multi-tenant private Kubernetes instances. The Deep Learning cluster of HPC4AI that is used for the DeepHealth project is composed of:

- 10 Compute nodes (currently being extended to 30): Intel 40 cores HT, 4xT4 Turing GPU per node, (1280 tensor cores), 10TB flash per node
- all-flash hyperconverged storage nodes: Intel 40 cores HT, 48TB flash per node (~200TB)
- 2x25 Gb/s bonded Ethernet

The overall software architecture is based on the open-source OpenStack cloud technology with multi-tenant configuration. Multi tenancy allows combining resource sharing through virtualization with resources and data isolation for each tenant. In HPC4AI, each OpenStack project can be configured as an HPC Secure Tenant (HST), that has a VPN with controlled access comprising an elastic configuration of private hosts, storage and network. Access to the HST through a VPN gateway guarantees the access to the overall environment in the tenant, either a Kubernetes cluster or a scheduling management system.

The OpenDeepHealth platform (shown in Figure 18) is the target infrastructure designed for the DeepHealth project and, is defined as an HST where a multi-tenant Kubernetes Container Cluster is deployed. This cluster is defined to specifically provide Docker containers integrating both ECVL and EDDLL libraries, for both CPU and GPU nodes (see also D4.1 deliverable).

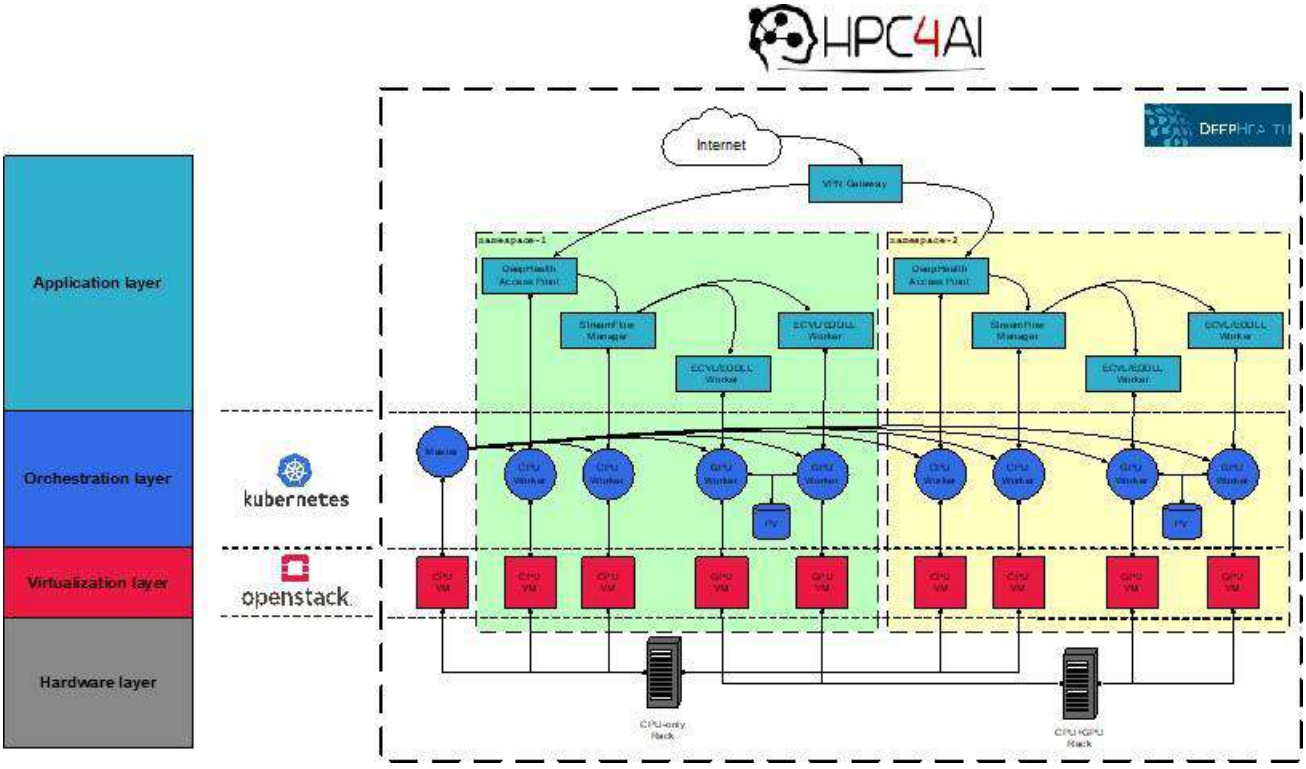

Figure 18. OpenDeepHealth platform.

An orchestration layer on top of Kubernetes cluster is provided by the StreamFlow workflow manager [18]. StreamFlow allows one to easily manage workflow modelling and execution in different environments, enhancing Kubernetes' ability to handle different computational steps. Also, StreamFlow makes it possible to describe a complex application as a workflow, and annotate it with an execution plan targeting different nodes, e.g. selecting GPU nodes when needed, and spawning across multiple sites, correctly, in this case, allowing access to OCCAM facility.

The idea behind this approach is that the ability to deal with hybrid workflows – i.e., to coordinate tasks running on different execution environments – can be a crucial aspect for performance

optimization when working with massive amounts of input data and different needs in computational steps. Accelerators like GPUs, and in turn different infrastructure like HPC and clouds, can be more efficiently used selecting for each application the execution plan that better fits the specific needs of the computational step of the ML applications developed in the project.

2.5.1 Planned Activities

The OpenDeepHealth platform is currently at an initial stage. Further activity is needed to complete the design and configure the HST. Also, the StreamFlow orchestration tool will be enhanced to provide a complete Deployment-as-a-Service technology, mainly based on configuration templates. The integrated environment comprising ECVL and EDDLL libraries will also be tested in a different configuration of the target infrastructure, primarily investigating the benefits of using GPUs as accelerator components.

3 Global runtime services and low-level runtime support

In this section we describe activities within WP5 addressing low-level runtime support and runtime services exploration, linked with activities in T5.4 (reported in D5.4).

3.1.1 Runtime Services Exploration

The COMP Superscalar (COMPSs) runtime (COMPSs documentation, version 2.6, n.d.) [14] is used in the project to efficiently distribute the computation of the DeepHealth libraries while hiding the infrastructure complexities. As described in D5.4 [15] (BSC, April 2020), it includes different infrastructures:

1. A set of distributed computing resources, where the workload of a single EDDL/ECVL operation is distributed as a monolithic service, in a Linux-based environment.
2. A hybrid cloud infrastructure, where the workload of a single EDDL/ECVL operation is distributed as a containerized service, i.e., based on Docker images.
3. An HPC-based infrastructure, where the Global Resource Manager (GRM) is used to instruct COMPSs when a set of EDDL/ECVL operations are distributed.

Deliverable 5.4 (BSC, April 2020) includes a complete description of these services, and a set of preliminary demonstrators that show the current status of development.

3.1.2 Low-level Runtime Management Policies

The first step towards the implementation of low-level runtime management policies is the integration of one of the use cases of DeepHealth into both the low-level libraries (EDDL/ECVL) and the upper layers of the stack. For this purpose, as a preliminary step towards the development of policies, EPFL has been working on porting the algorithms of UC13 to the EDDL. Details on the setup of UC13 working with the EDDL, as well as its integration with COMPSs and the GRM, are provided in D5.4. Overall, the current integration level consists in porting a convolutional neural network composed of 4 layers with convolution and maxpooling from Keras-Tensorflow to EDDL. Comparison between EDDL and Keras-Tensorflow were performed to generate accuracy comparison results. We also performed performance profiling on the EDDL implementation to find bottlenecks in the execution in CPU which could be improved by the use of runtime management policies, and find the adequate control parameters for low-level runtime management that could have an impact on performance of the application.

The detailed analysis and the policies implemented will be provided in the second version of this deliverable, in M25, by which time we will have enough data from multiple use cases to provide detailed results on the benefits low-level management policies.

4 First Analysis on EDDL/ECVL Support and Adaptations

In this section we describe the activities related with the first analysis and explorations of HPC systems and heterogeneous systems for the efficient execution of the project libraries. First, we delve into a report on the usage of the EDDL on the Marenostrum system targeting both CPUs and GPUs. Then, we focus on FPGAs and the opportunities offered by this technology to workloads based on the EDDL and ECVL libraries.

4.1 Initial Support for EDDL on Marenostrum System

We have performed a preliminary characterization of the EDDL library to identify the critical components of the library and to understand their computational requirements, providing the fundamental knowledge to guide the adaptation of EDDLL to heterogeneous HPC hardware (Task 2.3), as well as to provide valuable feedback to other ongoing tasks (e.g., Tasks 5.2, 5.3, and 2.4). Although many components of the library are still under development, we have placed our focus on detecting the most critical bottlenecks of the library and characterizing any performance issues. The resulting high-level characterization anticipates the main strategies of optimization required to exploit the heterogeneous HPC infrastructure efficiently.

4.1.1 Experimental Setup

As described in D1.2, the BSC 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. BSC hosts several HPC machines, being Marenostrum 4 the most relevant one. Marenostrum 4 consists of 48 racks with 3456 nodes of two Intel Xeon Platinum chips, each with 24 cores 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 peak performance of 11.15 PetaFLOP/s. The nodes are interconnected by a low-latency Omnipath network with a fully connected fat-tree topology.

Additionally, Marenostrum 4 is equipped with a cluster featuring emerging technologies that combines IBM POWER9 CPUs and NVIDIA Volta GPUs (V100). 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 similar to 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 of 1.57 PetaFLOP/s in double precision computations.

Several profiling tools have been used for the experimental evaluation and characterization of the EDDL library. Foremost, we have utilized the performance counters for Linux (PCL or perf) to access the hardware Performance Monitoring Counters (PMC), monitor specific kernel-based subsystem events, and collect high-level performance metrics like cycles and instructions executed per function. Also, the profiling tool Intel® VTune™ Amplifier has been used on the Xeon platform to support further characterization experiments. In this evaluation, the VTune profiler has facilitated tracking multithreading synchronization and scalability problems, capturing specific PMCs of the Intel Xeon processor that is being used, and also understanding interactions between the critical computing kernels within the library. Likewise, for the EDDL executions exploiting the GPU architecture, we have employed the NVIDIA Visual Profiler to obtain relevant information on the offloading and execution of the GPU computing kernels. Additionally, in order to model the memory usage and detect any issues and limitations related to the allocation/deallocation patterns, we have performed several analyses using the Valgrind memory profiling tool.

Notwithstanding the fact that the EDDL and the use cases are still under development, we have focused our analysis on the available EDDL prototype to characterize the main core functions of the library and broadly anticipate the most relevant performance issues and optimizations opportunities.

We will progressively incorporate into our analyses the different use cases as soon as they become available. We have selected the use case UC12 (Skin cancer melanoma detection) to drive our initial analyses. In this case, we have selected a representative subset of the ISIC 2019 challenge database assembled by randomly selecting 200 images for training, 10 for validation, and 10 for testing. Note that the primary goal of this task is to characterize the performance of the EDDL library and not to evaluate the accuracy of the methods implemented. Moreover, this task aims to focus on modelling the performance and scalability of the system. Hence, all the experiments have been executed for different image batch sizes and numbers of threads in each HPC platform. Following an iterative and incremental methodology, we will extend our initial findings with the feedback from WP4 (individual use cases) and WP5 (T5.1 and T5.2).

4.1.2 Experimental Results and High-Level Characterization

Following a top-down approach, we first evaluate the overall execution performance and memory consumption running on both hardware platforms (i.e., the Intel Xeon CPU and Volta100 GPU) for different batch sizes and working threads. Then, we narrow down the analysis to detect critical functions in the library and identify specific performance bottlenecks. Afterwards, we analyze the interrelation between compute kernels (i.e., computationally intensive functions) identifying the critical call-path between them. We conclude by compiling the most relevant results found during the analysis and experimentation. Moreover, we determine the major bottlenecks of the library that will be accelerated on Task 2.3. Furthermore, we outline the main optimization strategies proposed to this task (T2.3) to mitigate the inefficiencies that currently degrade the performance of the EDDL library on the HPC system.

4.1.2.1 Analysis of execution time, memory footprint, and scalability

For this analysis, Table 4 shows the running time and peak memory obtained for different executions of the UC12 on an Intel Xeon Platinum computing node. In general, CPU executions take from 9-20 seconds per sample image trained and approximately 600MB of memory per batch unit. The results show that execution times are slightly improved when employing more working threads and bigger batch sizes. Nonetheless, scalability rate is not proportional to the number of threads used and, therefore, suboptimal. In the same way, increasing the batch size only reaches a 1,22x speedup in the best case.

MN4 CPU		Threads											
		1		2		4		12		24		48	
		T	Mem	T	Mem	T	Mem	T	Mem	T	Mem	T	Mem
Batch Size	1	4127	3,1	3010	3,1	2433	3,2	2205	3,6	2151	4,1	2260	3,1
	5	3840	5,2	2711	5,2	2132	5,3	1897	5,7	1880	6,1	1933	5,2
	10	3816	7,8	2681	7,9	2084	8,0	1857	8,4	1807	8,7	1866	7,9
	20	3795	13,2	2649	13,2	2054	13,3	1820	13,6	1786	13,8	1816	13,2
	40	3772	23,8	2632	23,9	2040	23,9	1802	24,1	1763	24,1	1791	23,8

Table 4. Time performance (measured in seconds) and memory consumption (measured in GBytes) of the UC12 executions on an Intel Xeon Platinum varying number of threads and for different input batch sizes.

Despite not scaling properly with the number of working threads, the increase in memory required per each additional thread executed is negligible. Overall, the total memory footprint required is

proportional to the batch size configured. Consequently, the total memory installed on the computing node represents a practical upper limit on the maximum number of samples that can be processed in parallel (e.g., on the Intel Xeon Platinum nodes, equipped with 96GB of memory, a maximum of 160 samples can be processed in parallel).

GPU		Threads															
		1		2		4		12		24		48		90		160	
		T	Mem	T	Mem	T	Mem	T	Mem	T	Mem	T	Mem	T	Mem	T	Mem
Batch Size	1	111	1,5	110	1,5	111	1,5	116	1,5	111	1,5	111	1,5	111	1,5	111	1,5
	5	89	1,5	89	1,5	89	1,5	89	1,5	89	1,5	134	1,5	89	1,5	89	1,5
	10	86	1,5	86	1,5	86	1,5	87	1,5	86	1,5	86	1,5	86	1,5	86	1,5
	20	85	1,5	85	1,5	85	1,5	85	1,5	85	1,5	85	1,5	85	1,5	85	1,5

Table 5. Time performance (measured in seconds) and memory consumption (measured in GBytes) of the UC12 executions on a POWER9 computing node equipped with 4 Volta100 GPUs. The results show executions varying the number of working threads and image batch size.

Similarly, Table 5 shows the execution time and memory footprint for the GPU executions. As the results show, the configured number of threads has no significant effect on the overall execution time. Presumably, the offloading of critical computing kernels to the GPUs precludes the possibility of exploiting working thread parallelism. In contrast, increasing the batch size shows a moderate reduction in the execution time. In particular, increasing the batch size up to 20 can yield a speedup of 1,30x in the best case. In this way, GPU executions take ~0.5 seconds per sample image trained, achieving 18-40x speedup compared to the CPU executions.

Overall, these results indicate that the executions on both CPU and GPU platforms do not fully exploit all the computing resources available. In this way, EDDLL executions scale inadequately with an increasing number of working threads. Moreover, selecting larger batch sizes, as to increase parallel processing opportunities, only yield minor performance improvements. Taking into account that the memory requirements remain invariable to all the executions, these results suggest possible serialization of critical computations, data dependency problems or synchronization issues.

4.1.2.2 Analysis of critical functions and performance bottlenecks

Intending to identify the most critical functions of the EDDLL, Table 6 shows a relation of the most time-consuming functions and modules obtained from the execution of the UC12 on an Intel Xeon node.

Function	Time	%	CPI	Module
func@0x18810	591.6s	33,8%	20.2	OpenMP
Eigen::internal::general_matrix_matrix_product	472.5s	27,0%	0.3	Eigen
Eigen::internal::general_matrix_matrix_product	263.6s	15,0%	0.3	Eigen
Eigen::internal::gebp_kernel::operator()	145.6s	8,3%	0.3	Eigen
func@0x189a0	126.8s	7,2%	19.8	OpenMP
im2col	40.8s	2,3%	0.7	EDDLL

Eigen::internal::general_matrix_matrix_product	27.7s	1,6%	0.3	Eigen
[Outside any known module]	22.3s	1,3%	8	N/A
get_pixel	16.5s	0,9%	1.1	EDDLL
add_pixel	10.5s	0,6%	0.7	EDDLL
Eigen_internal_gemm_pack_const_blas_data_map	9.5s	0,5%	4.4	Eigen
Eigen::internal::gemm_pack_lhs::operator()	6.8s	0,4%	3.3	Eigen
cpu_add._omp_fn.33	6.5s	0,4%	0.2	EDDLL
ecvl::RearrangeChannels	2.7s	0,2%	0.3	ECVL
Eigen::internal::general_matrix_vector_product	2.2s	0,1%	1.2	Eigen
Eigen::internal::call_dense_assignment_loop	2.0s	0,1%	1.4	Eigen
Eigen::internal::general_matrix_vector_product	1.8s	0,1%	1.3	Eigen
fast_randn	1.6s	0,1%	0.4	EDDLL
cpu_mpool2D	1.1s	0,1%	0.4	EDDLL

Table 6. Profile summary of the most time-consuming functions for the UC12 execution on an Intel Xeon node. The execution was performed using a reduced dataset composed by 10 images, for 2 epochs, using 48 threads. For each function, the table displays total CPU-time, percentage of time (with respect to the total CPU time taken by the execution), cycles per instruction (CPI), and the module or framework each function belongs to (i.e., EDDLL, ECVL, OpenMP, or Eigen).

As the table reflects, more than 40% of the time is spent on synchronization functions related to the OpenMP framework. In effect, these functions correspond to waiting methods whose main purpose is to synchronize working threads spawned by the OpenMP runtime; hence, the extremely elevated CPI of these functions.

Besides the synchronization overhead, more than 50% of the total CPU time is devoted to matrix computations within the Eigen library. This linear algebra library offers highly-optimized matrix/vector arithmetic operations. In the case of the EDDLL, it supports core operations between layers of the DNN. Nevertheless, Eigen's efficient implementation archives a remarkable CPI of 0.3, which is near the theoretical best CPI of modern superscalar processors (i.e., around 0.25). Furthermore, this result suggests that the core operations of the EDDLL are conveniently exploiting the hardware resources of the architecture. In turn, optimizations targeting these core functions should aim to reduce the overall number of executed instructions or improve code vectorization using wider SIMD instructions.

Finally, note that only 4.4% and 0.2% of the CPU time is spent on EDDLL and ECVL support functions, respectively. The CPI of these functions indicates that though it is possible they may be improved, their weight in the overall execution time entails that such work would not result in a significant impact on the performance of the library.

In general, these results indicate that the main bottleneck of the EDDLL is related to synchronization issues between working threads that, in turn, prevents the proper scalability of the system. Besides, most computing-intensive functions perform simple and well-defined matrix operations that not only have been extensively analyzed and optimized but also can be effectively offloaded to hardware accelerators. Indeed, these matrix operations depict traditional data parallelism that can be exploited by hardware devices such as GPUs and FPGAs. Even for CPU-only executions, these results suggest that the library performance can largely benefit from fully exploiting the vectorial capabilities of current HPC platforms.

4.1.2.3 Analysis of the call graph and critical kernels on the critical path

Considering the results presented in Table 6, we can assess that the most computational intensive functions involve performing matrix operations with the layers of the DNN. In particular, those operations lie at the core of the forward and backward propagation functions during the training of the DNN. Unsurprisingly, these functions represent the most computing-critical components of every modern DL framework. In the case of the EDDL library, these functions represent more than 99% of the wall-clock time devoted to training the model (see Figure 19).

Figure 19. Timeline depicting the execution time spent by each training batch. The figure breaks down the training process displaying the time spent on each of the functions called by the train_batch_t method.

More in detail, Figure 20 summarises the call graph from the high-level method train_batch_t down to the cpu_conv2D function in charge of performing the 2D convolution for the forward propagation. From the operational point of view, the cpu_conv2D function calls Eigen routines to perform a series of matrix operations which take up around 40%. In the same way, the remaining 60% of the time is spent executing the kernels cpu_conv2D_grad and cpu_conv2D_back for the backward propagation.

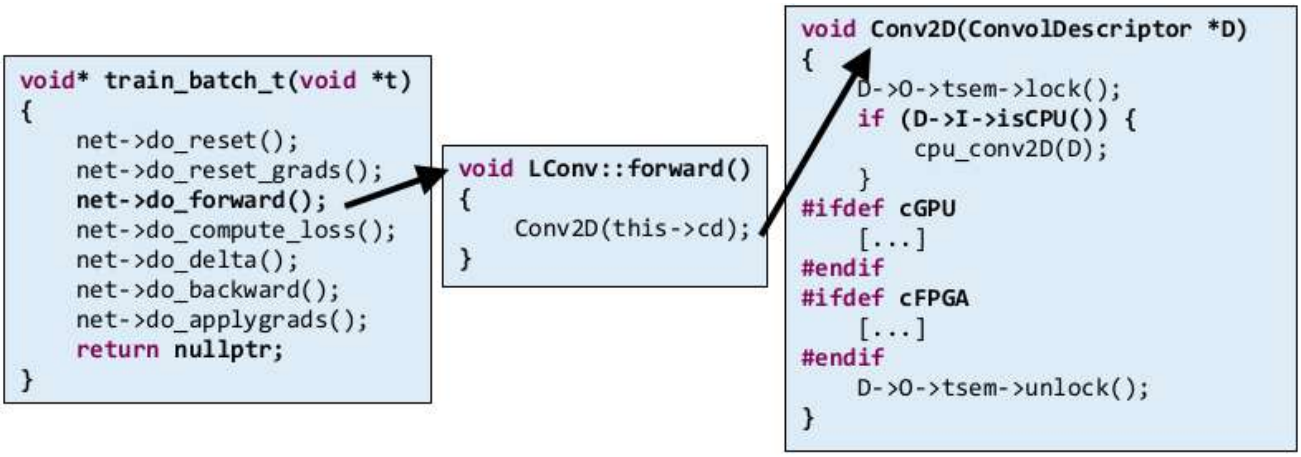

Figure 20. Simplified pseudocode of the train_batch_t method and the chain of calls from do_forward (forward propagation function) down to cpu_conv2D (DNN 2D convolution).

Altogether, almost the totality of the wall-clock time spent to train the DNN model is invested in executing the low-level EDDLL functions cpu_conv2D, cpu_conv2D_grad, and cpu_conv2D_back (critical computing kernels of the EDDLL). These functions essentially perform matrix additions and multiplications through calls to Eigen routines. Moreover, each of these functions operate over a single batch in mutual exclusion from other batches or epochs of training. Even so, computations of the whole batch (and even within each batch) can be performed in parallel or be offloaded to a hardware accelerator. Actually, the current EDDLL implementation already contains switches for offloading these computations to GPUs and FPGAs, along with the regular CPU implementation (Figure 20).

However, computing each batch in mutual exclusion represents a potential bottleneck as the training process is forced to serialize computing work between different batches and epochs. Moreover, offloading computations entails certain overheads (e.g., transfers to/from the host, resource allocation, or configuration of the device) and synchronization penalties. For those reasons, it is paramount that the computation payload provided to each of these critical kernels is large enough to maximize the utilization of the hardware resources available on each platform. Otherwise, the EDDLL library executions can incur in severe synchronization penalties and

slowdowns due to thread overhead that, ultimately, would render the available computing cores underutilized.

4.1.2.4 Summary of the Main Characterization Results

Overall, executions using the EDDLL library are computationally bound and may incur in large memory requirements depending on the batch size configured. In particular, the most computationally intensive kernels of the library are dedicated to the forward and backward propagation during the training of the DNN. These critical kernels perform relatively simple matrix operations using Eigen's linear algebra support functions. They can be easily parallelized and offloaded to hardware accelerators.

In the current EDDLL implementation, the principal performance bottleneck of the library involves synchronization penalties between working threads performing core operations during the DNN model training. These waiting times and synchronization inefficiencies prevent proper performance scalability of the system with an increasing number of working threads. For this reason, optimization efforts should address these synchronization issues in order to maximize the hardware resource utilization. In the same way, the previously identified computing kernels are bound to be accelerated by means of offloading them to GPU and FPGA devices. Ultimately, these hardware accelerators will allow the exploitation of the parallelism depicted by these critical functions and effectively improve the overall performance of the EDDLL.

4.1.3 Planned Activities

The remaining work will focus on addressing synchronization issues in order to maximize the hardware resource utilization. In the same way, we plan to address the acceleration of the previously identified computing kernels by means of offloading them to GPU and FPGA devices. Further characterization and adaptation efforts will be focused on evaluating the performance of the offloaded kernels, as well as the overheads derived from data transfers between host and accelerator devices as well as device configuration. This way, the methodology and optimization guidelines presented here will be also used towards analysis of the ECVL in the context of WP3. In turn, these analysis and optimization results will provide valuable feedback for further developments in WP2.

4.2 Initial Support for ECVL/EDDL libraries on FPGA devices

To support the acceleration of the ECVL and EDDL libraries, FPGA devices are used to perform the main computational kernels inside the two libraries. The adopted strategy is resumed in Figure 21.

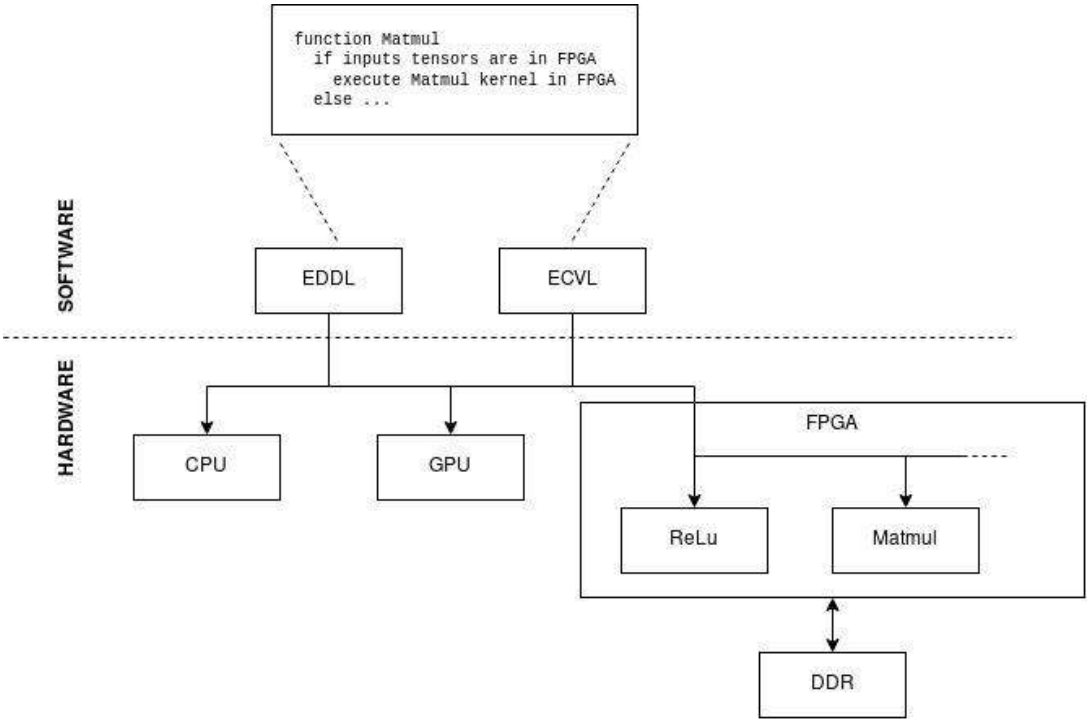

Figure 21. Adaptation of FPGA kernels into EDDL/ECVL libraries.

Each of the kernels that are going to be accelerated has a dedicated portion of the FPGA physical resources. The input and output tensors are placed inside the local DDR memory to avoid costly memory transfers between the host memory and the device memory. The host is responsible for orchestrating the execution of the kernels: the FPGA acts as a passive device, waiting for commands and notifying their completion. Kernels can be replicated multiple times to support parallel execution, if the needed resources are available on the device. If the needed computation is known beforehand, the set of kernels placed on the FPGA can be tailored around the use case to make a more efficient use of the resources. On the other hand, once an FPGA configuration is generated, it can be reused to accelerate many different use cases as long as the basic operations have been synthesized on the FPGA.

Figure 22. Alveo FPGA from Xilinx used as test device.

The identified strategy has been implemented on an Alveo U200 FPGA board from Xilinx (Figure 22). Using this FPGA allowed starting EDDL and ECVL adaptations in parallel with the development

of the MANGO cluster support since they share the same programming model. The Alveo board has been designed for HPC applications in data centers, being pluggable in a standard PCI-Express extension slot on both commodity desktop and server systems. The FPGA resources are split in two parts: the static and the dynamic part. The static part contains the infrastructure to allow the user design to communicate efficiently with the host system and the DDR memories placed on the board. The dynamic part, that is the majority of the FPGA, is reserved for the user design, and can be quickly reprogrammed at runtime.

To ease programmability and shorten the development time, it supports two complete High Level Synthesis flows: one based on the standard C++ language with custom compile-time directives (pragmas) to drive the synthesis process, and another one based on the Khronos OpenCL standard. These two options provide a fast way to synthesize an FPGA design while working at an abstraction level significantly higher than standard EDA languages, notably VHDL and SystemVerilog. Moreover, different emulation flows are provided to be able to eagerly test and validate the code before going through a full FPGA synthesis flow, which could take up to several hours.

Communication from the host to the FPGA is implemented through a stack of software libraries provided by the vendor. On top of this stack we find an OpenCL API that enables the interaction between the user application and the memories and FPGA placed on the board. In this way, the host application is decoupled from the specifics of the hardware implementation: the code is portable across all the FPGAs supported by the tool flow. The OpenCL API interacts with the underlying software stack until it reaches the kernel driver, which is in charge of handling the physical communication through the PCI-Express bus.

The software tools to program the Alveo card are collected in a framework called Vitis. Leveraging the functionalities of Vitis, the kernels in the EDDL and ECVL libraries have been implemented in HLS.

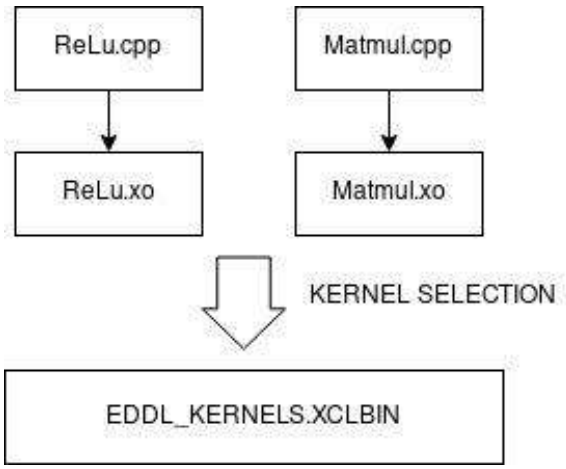

Figure 23. Kernel instantiation in EDDL/ECVL libraries.

The kernels are defined and compiled separately, thus, they are assembled together taking into account the workload that is going to be run on FPGA. First of all, the kernels are generated using the software emulation mode. This mode is used to generate a functional model of the compiled kernel, which roughly approximates the hardware implementation and allows validating the code before proceeding with the more time-consuming hardware implementation. The second step is to proceed with the hardware emulation flow. The hardware emulation generates a synthesized RTL model, which can be used to perform cycle-accurate simulations of the kernel, including also a good approximation of the host-to-FPGA communication pattern. When used with the optional profiling flags, it allows identifying bottleneck and optimization opportunities early in the design phase, which is very handy when a performance analysis is required. The last step is the hardware generation

flow, which takes the kernel code and fully implements it on the target device, with the output being a bitstream file ready to be flashed on the reprogrammable part of the FPGA.

4.2.1 EDDL library

The EDDL library offers an API to define and run a specific neural network on a target device. The main core of the library is the Tensor concept, which is implemented as a class and offers a wide range of manipulations. The Tensor interface is device independent, thus enabling a strong decoupling between the network training logic and the hardware implementation of the required operations.

First of all, the compilation scripts of the library have been modified to add an additional flag that can be used to compile the library with FPGA support. The compute service abstraction is offered to the library users to select onto which device training should be run. The FPGA compute service can thus be selected successfully when the library has been compiled with FPGA support.

Model definition is the only place where the user must be aware of the underlying hardware platform: the model is copied to the target device when the model is built. All the implementation details are hidden away through the Tensor abstraction. Indeed the Tensor class interface offers the chance to run a wide range of tensor manipulations using an API that is device-agnostic. The class implementation keeps track of the physical device where the tensor is placed, allowing operations to be dispatched to the correct device implementation. As we are relying on the OpenCL API to implement host-to-FPGA communication, we extended the Tensor class to add a handle to the OpenCL buffer representing the tensor memory area in the device memory.

We proceeded with the implementation of the kernels used in one of the many EDDL examples. The example trains a simple neural network on the MNIST dataset. This example allows identifying the most basic kernels needed to run a complete training. We therefore report the kernels list:

Kernel	Description
Add	Adds two tensors of the same size
Sum	Sums all the elements of a tensor
Sum2d row-wise	Adds a 1D tensor to each row of a 2D tensor
Reduce sum2d	Reduces a 2D tensor to a 1D tensor, summing all the elements along one direction
Relu / Softmax	Various activation functions used in neural networks
Mult2d	Matrix multiplication
D_Relu	Derivative of the relu activation function
Cent	Cross-entropy loss function implementation
Accuracy	Compares network outputs with known labels, count the correctly classified images in the batch

The FPGA implementation of the mentioned kernels is specified in Vivado HLS language, which is an extended version of the C++ programming language. The kernels take one or multiple buffers as input and output, along with any additional parameter. For each input or output buffer, a standard AXI port is generated to ease the exploitation of parallelization opportunities. Scalar arguments are passed through a single AXI Lite interface, which is the same one used by the Xilinx runtime to launch the kernels and query their status.

Then, we extended the EDDL compilation flow to support the integration with the Vitis toolflow, used to generate a bitstream file containing all the kernels that could be simulated or flashed on the physical FPGA.

A special focus has been put on the matrix multiplication operation, as it is used to implement both Fully Connected and Convolutional layers in neural networks, and it is currently the main computation bottleneck in these workloads. We therefore describe next the several options we considered.

4.2.1.1 RTL-based Systolic Architectures

Systolic architectures are a well-known approach to implement hardware accelerators for algebraic manipulations, like the matrix multiplication. They are composed of a grid of processing elements where data flows simultaneously in two spatial directions. The processing element implements a simple multiply-and-accumulate, storing locally the result of the operation and propagating it outside at the end of the execution. The grid can be equipped with external buffers to account for the fact that matrices can be greater than the grid, in that case additional logic is required to inject and eject the data with the correct timing order.

The grid structure can be easily adapted to the internal structure of FPGAs, where compute elements are placed in DSP slices and distributed over the chip area (see Figure 24), also providing routing resources to interconnect them at will.

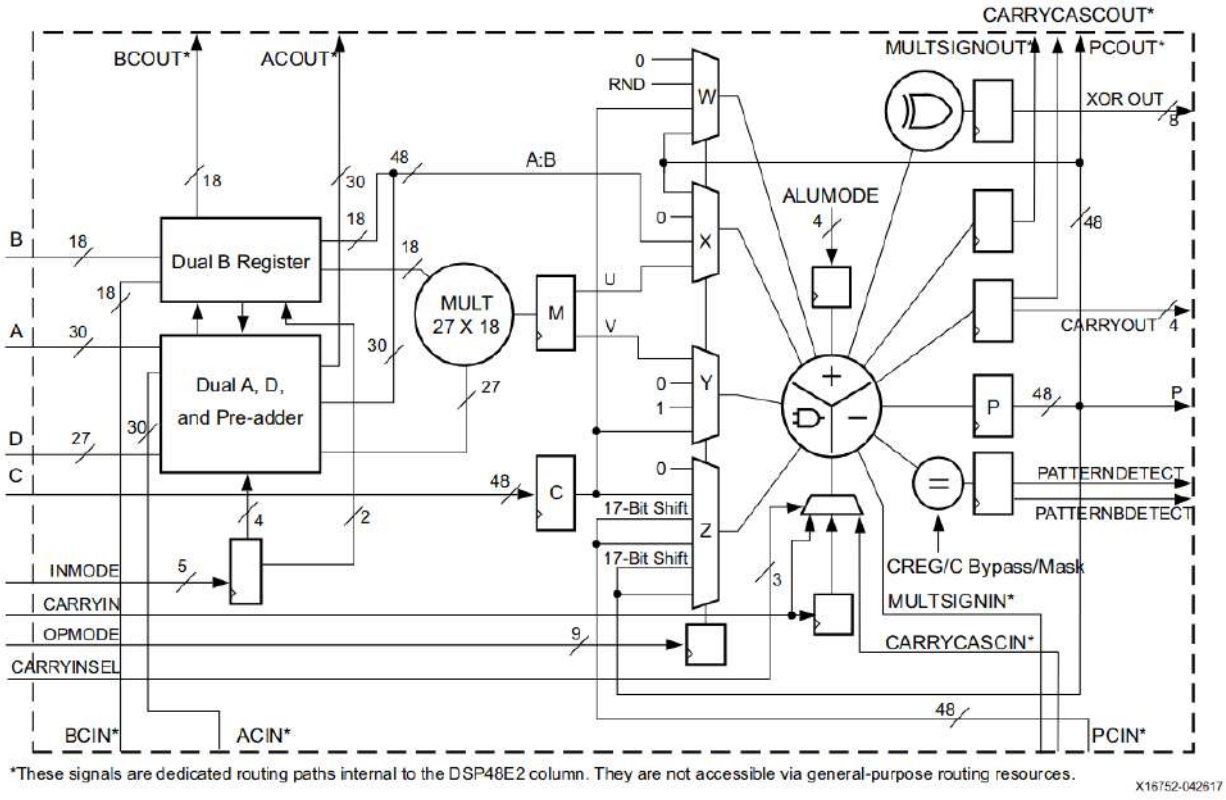

Figure 24. FPGA DSP basic block.

In Figure 24 we can see an example of a DSP slice used in the Alveo FPGAs. A DSP slice internally provides all the functionality required to implement a full multiply-and-accumulate operator. However, it should be noted that the supported computation must be performed in 16 bits, fixed-precision. When a need for a greater precision arises, additional logic must be placed to combine together multiple DSPs to form a wider precision Multiply-and-Accumulate operator.

Although an optimal solution, systolic accelerators cannot be easily implemented on HLS flows. Indeed, the HLS compilation flows take charge of inferring the optimal timing of the circuit, and do not leave many opportunities to infer such a detailed internal structure. So, if a systolic accelerator is desired, development should fall back to RTL code, with all the pitfalls it carries, like longer development cycles and a significantly greater engineering effort.

4.2.1.2 Xilinx GEMX

The GEMX IP core is an open source component provided by Xilinx to accelerate algebra-heavy computational workloads. The IP core is released in source form, ready to be synthesized with the

Vitis tool flow as a stand-alone project or integrated with several other kernels. For our use case, we integrated the IP core with the other kernels developed at UPV.

The IP core is basically an instruction processor, which takes complex instruction descriptors representing one of the several operations supported: matrix-matrix multiplications, matrix-vector multiplications, transpositions, sparse matrix operations, and more complex functions implementing domain-specific operations. The interfaces of these functions are modelled on the standard BLAS interfaces, to ease the transition of existing codebases.

Moreover, an extensive set of parameters is provided, such that the behaviour of the component can be tailored around the specific application needs. An accurate selection of the mentioned parameters can result in a significantly smaller core size, thus freeing up resources for the other kernels. The main parameter for deep learning applications is the compute precision, which can be arbitrarily chosen between any of the standard C numeric types, that is, 1, 2, 4, or 8-byte wide precision. The computation will eventually be mapped on the physical DSP resources available on the FPGA, possibly breaking the computation in pieces to account for the reduced precision of the DSP slices: the synthesis tool takes care of optimally performing this task. Floating-point types deserve special attention as there are no physical resources on the FPGA that are natively able to compute with them. The synthesis tool will thus infer the use of a mix of DSP slices and reprogrammable logic, significantly increasing the latency and the area of the core. In conclusion, different neural networks have different precision requirements, and depending on the network that is meant to be accelerated, the precision of the computation offers interesting optimization opportunities.

Notwithstanding the wide customization opportunities offered by the GEMX, we identified one main pitfall that could represent a roadblock in the adoption of the GEMX in the EDDL library. To let the synthesis tools infer efficient memory interfaces with the external DDRs, and also to reduce the complexity of internal control logic, the matrix sizes are strongly constrained. Indeed, each dimension of the matrices must be a multiple of a given factor, which can be calculated from a few IP core parameters. Even if it were theoretically possible to overcome this limitation by appropriately setting these performance parameters, the resulting parallelism level and memory bandwidth would compromise the performance to unacceptable levels.

The GEMX has been integrated in the EDDL as a mean to evaluate its performance and capabilities. The host API provided by Xilinx abstracts away the physical details of how instructions are represented, and the required memory layout. We integrated the API provided by Xilinx with the EDDL code responsible for launching the matrix multiplication on FPGA. From that point, we have performed an extensive profiling of the GEMX core for different matrix sizes, taking as a driving principle the characteristics of state-of-the-art neural networks. By comparing these performance levels with the original performance of the EDDL running on the CPU, we can draw the first considerations regarding the candidacy of the GEMX as a matrix-multiplication accelerator in the EDDL.

Figure 25 shows the performance analysis obtained for different devices, namely CPU and FPGA, adopting the same floating-point representation. On the x-axis different matrix sizes are shown and on the y-axis the average time obtained for multiple executions of the operation.

If we compare CPU and FPGA devices both in float type we see that for smaller matrix sizes the FPGA is slower. This is mainly due to the cost of initialization of the device and data transfer overhead. For larger array sizes (2048x2048) the execution time is considerably reduced even when keeping the floating point precision. In these cases, the execution time in matrix multiplication could be reduced by around 20% w.r.t the CPU.

As planned activities, in addition to the typical data type (float) in neural networks, time results will be explored for other data types, mainly shorts. The loss of precision resulting from other types of data will also be studied.

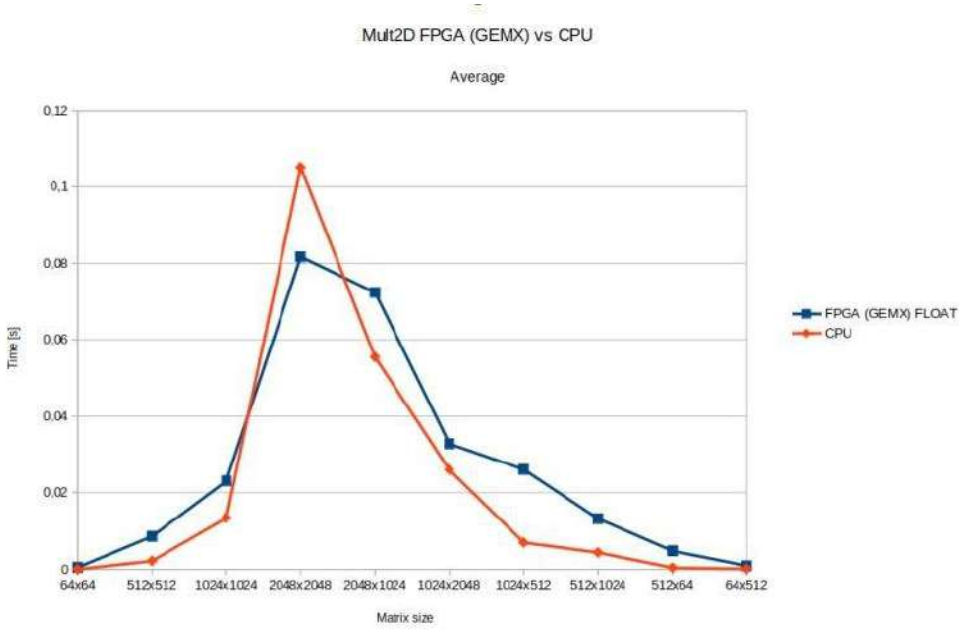

Figure 25. Runtime for 2D multiplication in CPU and FPGA (GEMX) on EDDL Library.

4.2.1.3 Custom HLS Matrix Multiplication Kernels

Besides the solutions provided by the vendor, another viable option is the development of a custom HLS kernel for the matrix multiplication acceleration. We found and evaluated a few open source kernels that are freely available and could be integrated inside the project.

One example is provided by Xilinx and freely available in their sample designs [20]. The kernel is written targeting the Vitis design flow and leverages several of the compile-time optimization opportunities presented by Vitis, in the form of pragmas. A matrix multiplication is an example of an application where accurate use of the pragmas is critical to achieve acceptable performance. Although the kernel is optimized, it can only be used to accelerate matrix multiplications of fixed sizes. This makes it a bad candidate for the EDDL acceleration, as the size of the matrices is not known at synthesis time and would require a different IP instance for each of the required sizes.

Another example is a GEMM IP core provided by SPCL and available on GitHub. The design aims to reproduce a complete systolic structure using just C code enhanced with the pragmas offered by Vitis to drive the synthesis process. However, driving the synthesis process with such a fine grain can be a difficult task, and that affects the maintainability of the source code. Moreover, as the tools are fairly recent and still immature, the code has to also take into account the specific characteristics (and problems) of the synthesis tools. The resulting code requires a significant maintenance effort and is not portable to other HLS synthesis tools. Similarly to other HLS approaches, the grid size poses constraints on the valid input matrix size, and there is a trade-off between performance and input matrix size restrictions.

As planned activity, we reserve the chance to explore in the future more custom approaches to develop a matrix multiplication kernel that presents both good maintainability and performance, while offering the required flexibility on the input matrix size.

4.2.2 ECVL library

Similarly to the EDDL, the ECVL offers a device-agnostic API. The main interface offered is that of the ECVL Image, which is an opaque representation of an image located in one of the compute devices (e.g., CPU or FPGA). Methods that operate on ECVL Images will check the device where the image is located, and execute the transformation there.

The tools used to develop the functions on the FPGA are provided by Xilinx XfOpenCV library. This library provides a software interface for computer vision functions accelerated on an FPGA device. The advantage with this library resides in the fact that their functions are mostly similar in functionality to their OpenCV equivalent, apart from some documented deviations. As the CPU implementation of the EDDL relies on OpenCV, it is ideal from our point of view to use a platform that keeps a similar interface.

XfOpenCV offers a set of kernels that are mainly thought for integration inside the Zynq boards, which are other platforms offered by Xilinx, featuring an on-chip CPU beside the FPGA reprogrammable fabric. In our case, as we are targeting Alveo boards, we had to write a thin wrapper layer that converted the xfOpenCV functions into Vitis kernels so that we could apply the acceleration strategy described at the beginning of this chapter. Moreover, we extended the ECVL library to support FPGA initialization, and we extended the compilation scripts to support the specification of the FPGA build target.

XfOpenCV presents some characteristics that are important to mention, and that we had to take into account when we developed the wrapper layer. The main one is that, while standard OpenCV C++ API functions take normal parameters, the XfOpenCV API functions convert some of these normal parameters into template parameters. As these template parameters need to be known at hardware generation time, the generated bitstream file will not support all combinations of parameters that are supported in CPU. For this reason, in the future we will need to carefully examine the requirements of the project use cases and identify which values of these parameters are required. Nevertheless, this lack of flexibility is a well-known drawback of FPGAs. Another characteristic of XfOpenCV functions is that input images are passed as xf::Mat instances, which are images of a fixed size. As we want to support different image sizes with the same hardware configuration, we call the XfOpenCV functions using a worst-case image size, thus generating the acceleration hardware for the biggest image size that we want to support. Then, in the wrapper layer we zero-pad the input image, bringing it to the maximum size and making it supported by the accelerator.

We proceeded with an initial comparison of performance numbers between the CPU and FPGA implementations of some image processing transformations. The general flow of operation is described below:

1. Read the input image using the ECVL API.
2. If we should launch on FPGA, copy the input image to the device.
3. Take a first timestamp.
4. Call the processing function, in CPU or FPGA depending on the configuration. Wait until completion.
5. Take a second timestamp. Record the execution time.
6. If we launched on FPGA, copy the output data back to the CPU.
7. Write the output image to disk using the ECVL API.

We have now implemented four wrappers for as many XfOpenCV functions: xf::resize, xf::Threshold, xf::GaussianFilter and xf::OtsuThreshold. We also noticed that the ECVL splits the functionalities offered by the xf::resize kernel in two functions, one dedicated to downscaling and the other one to upscaling. When making the tests, we made sure to cover both cases.

Now, we present some results from the functions mentioned before. We use three different images of different size and shape: Image 1 (600x900 RGB), Image 2 (2048x2048 RGB) and Image 3 (3072x2048 RGB). With the infrastructure we developed we are able to accelerate five ECVL functions:

1. Resize Scale: is the method used to resize the source image to the size of the destination image (in this case to a bigger size). Different types of interpolation techniques can be used in resize function, namely: Nearest-neighbor, Bilinear, and Area interpolation.
2. Otsu Threshold: it is used to automatically perform clustering-based image thresholding or the reduction of a grey-level image to a binary image. The algorithm assumes that the image

contains two classes of pixels following bi-modal histogram (foreground pixels and background pixels). Then, it calculates the optimum threshold separating the two classes. Otsu method is used to find the threshold that can minimize the intra class variance which separates two classes defined by weighted sum of variances of two classes. In this case, the ECVL CPU implementation does not rely on OpenCV's implementation and implements its own algorithm.

3. Threshold: it performs thresholding operation on the input image. The threshold value can be an arbitrary value (for example, the result of the Otsu Threshold function).
4. Resize Dim: it uses the same kernel as Resize Scale, but in this case it reduces the input image to a smaller size.
5. Gaussian Filter: it applies a Gaussian blur filter on the input image. Gaussian filtering is done by convolving each point in the input image with a Gaussian kernel.

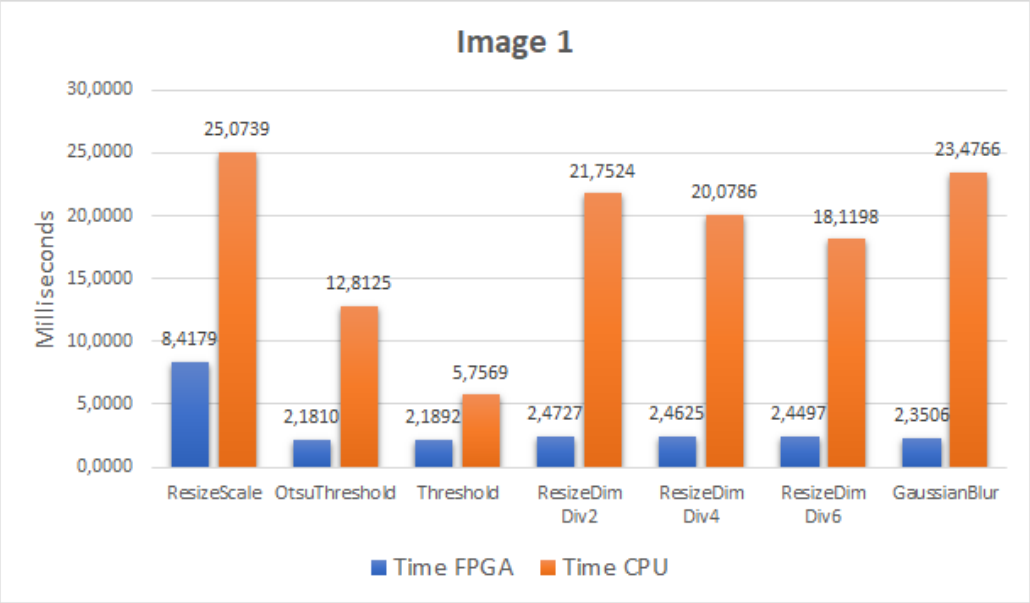

Figure 26. Execution time for different OpenCV kernels on both FPGA and CPU.

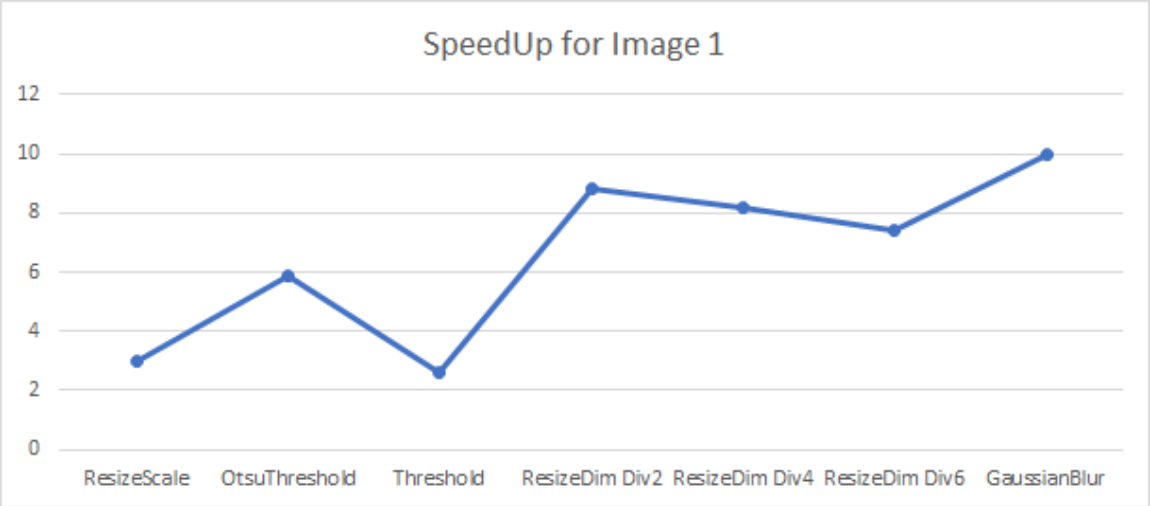

Figure 27. Speedup for different OpenCV kernels when running on FPGA vs CPU.

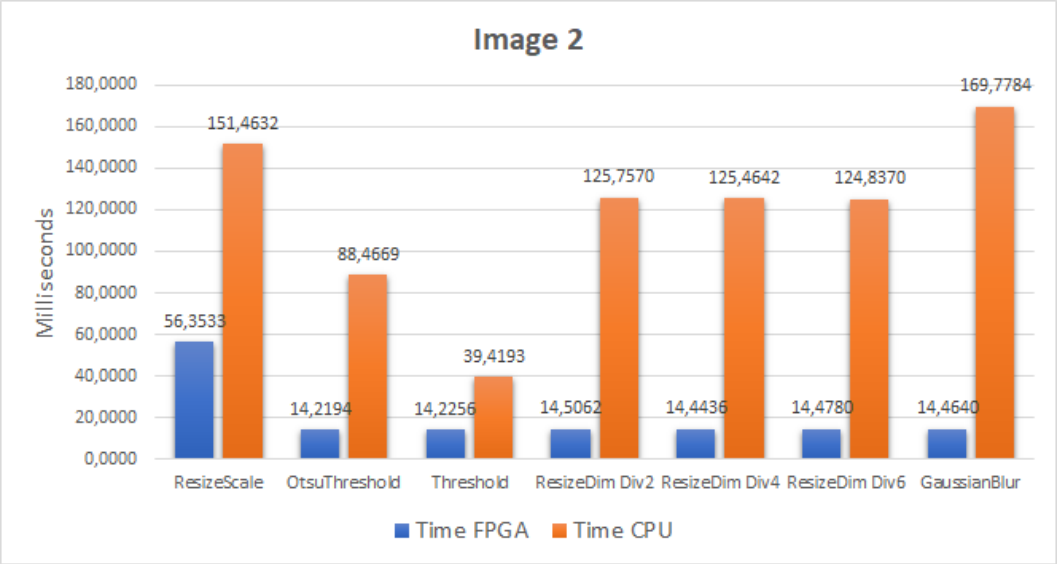

Figure 28. Execution time for different OpenCV kernels on both FPGA and CPU.

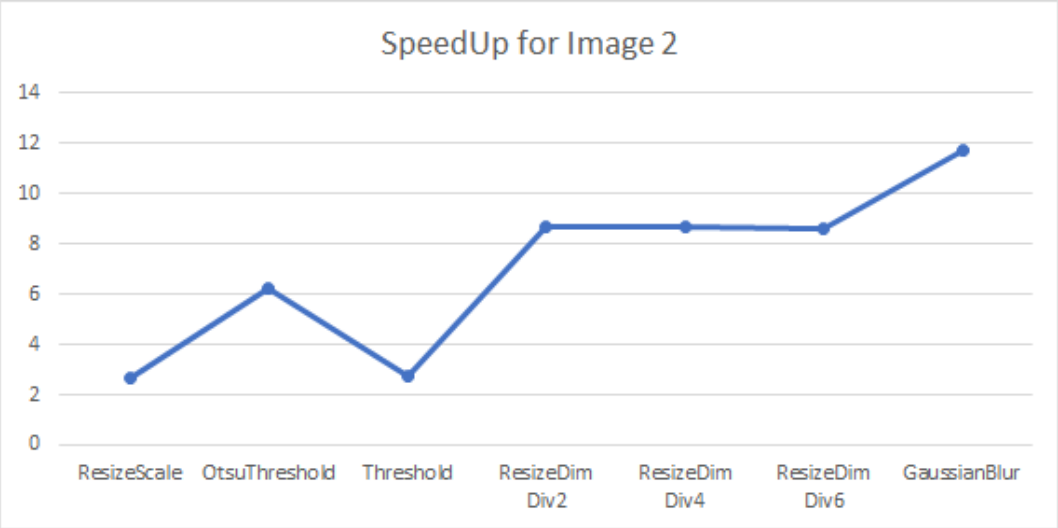

Figure 29. Speedup for different OpenCV kernels when running on FPGA vs CPU

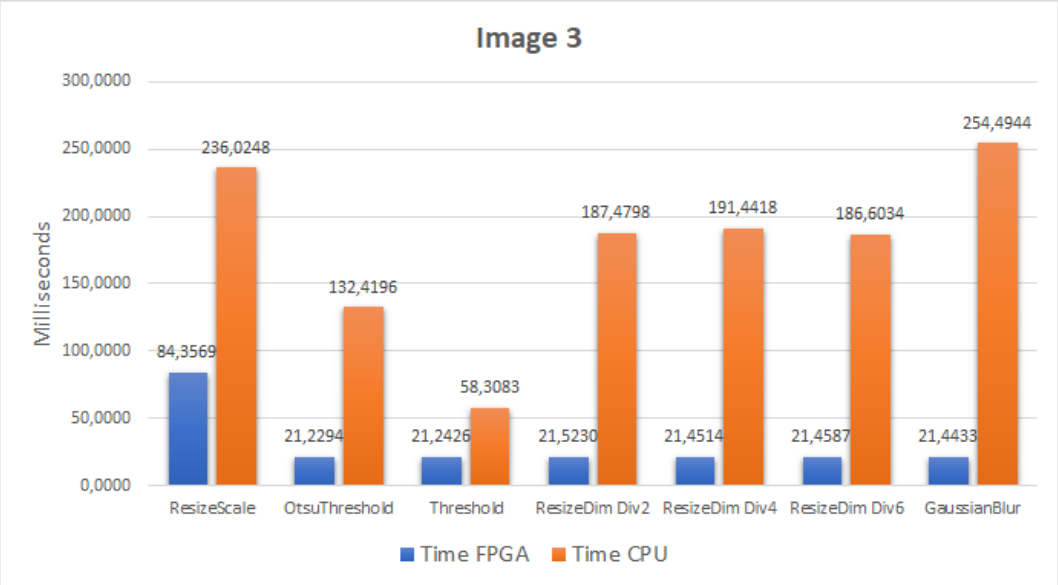

Figure 30. . Execution time for different OpenCV kernels on both FPGA and CPU.

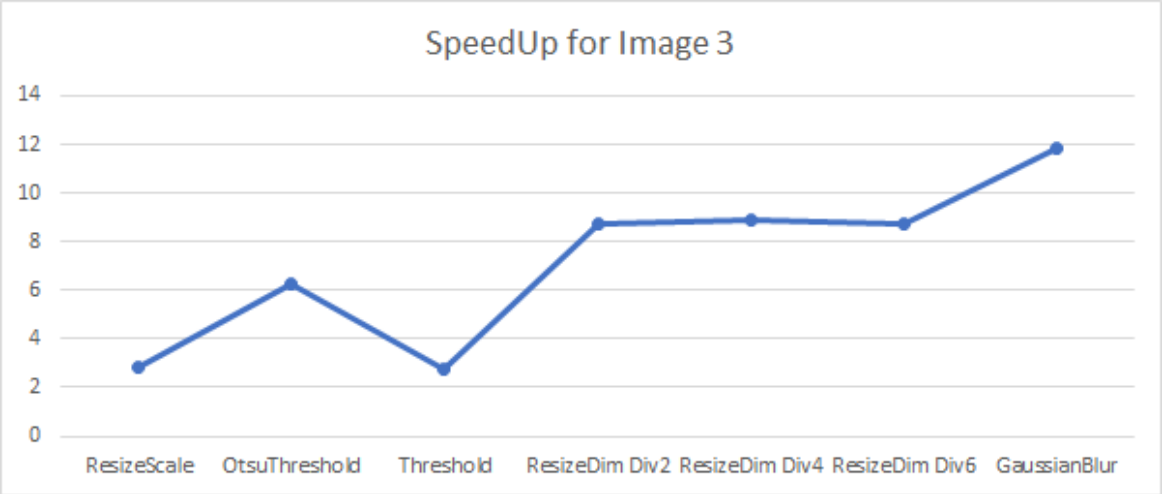

Figure 31. Speedup for different OpenCV kernels when running on FPGA vs CPU

As we can see for the results (Figure 26 to Figure 31), each transformation faces a significant speedup when run on FPGA, and that speedup is the same regardless of the size of the input image. It is also interesting to see that the ResizeScale and ResizeDim functions, although using the same underlying accelerator, face a very different speedup. For that reason we decided to also analyse the performance of the ResizeDim function changing between different reduction factors (2, 4 and 6 times the size of the original image). However, there is small variation between the three measures, indicating that the performance of this kernel is not affected by the scaling factor.

From this first performance result, we can conclude that the XfOpenCV set of kernels can be a good candidate to implement hardware acceleration of the ECVL library.

4.2.3 Planned Activities

The different options for matrix multiplications and for OpenCV kernels have been almost completed. The remaining work will focus on a final assessment of such possibilities together with the exploration of OpenCL-based kernels for matrix multiplication operations.

5 Conclusions

This deliverable reports activities performed up to M15 related to tasks T5.1, T5.2 and T5.3. The goal of such tasks is to deploy, analyse, and decide which are the most suitable technologies and strategies to use when porting the EDDL and ECVL libraries to heterogeneous computing technologies. The current deliverable will be updated in the beginning of the third year and completed towards the end of the project.

Currently, most of the focus has been put on the adaptations of FPGA devices and their possibilities both in hardware and in runtime for the project. Most effort has been devoted to the MANGO prototype and its optimization and customization. The low-level runtime is being adapted and the prototype has been finalized. This deliverable includes also initial analysis of EDDL performance when targeting both CPUs and GPUs in BSC on-site premises.

Initial tests on FPGAs for both EDDL and ECVL have been performed on the Alveo board, identifying key kernels to be adapted and customized for the proper implementation of both libraries on heterogeneous devices.

6 References

[1] J.Flich,G.Agosta,P.Ampletzer,D.A.Alonso,A.Cilardo,W.Fornaciari,M.Ko- vac, F. Roudet, and D. Zoni, “The MANGO FET-HPC Project: An overview,” in IEEE 18th Int’l Conf on Computational Science and Engineering (CSE). IEEE, 2015, pp. 351–354.

[2] http://www.supermicro.com.tw/products/SuperBlade/module/SBI-7128RG-F.cfm

[3] https://www.profpga.com/products/systems-overview

[4] https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf

[5] https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf

[6] https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf

[7] https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/s10_datasheet.pdf

[8] https://www.intel.com/content/www/us/en/programmable/solutions/acceleration-hub/overview.html

[9] https://www.xilinx.com/products/boards-and-kits/accelerator-cards.html

[10] https://www.khronos.org/opencl/

[11] P. Coussy and A. Morawiec, Eds., “High-Level Synthesis,” 2008.

[12] https://www.xilinx.com/products/design-tools/vivado.html

[13] https://www.intel.com/content/www/us/en/programmable/downloads/download-center.html

[14] “COMPSs documentation, version 2.6,” BSC, [Online]. Available: https://compss-doc.readthedocs.io/en/2.6/.

[15] BSC, “D5.4 The runtime system for DeepHealth libraries,” DeepHealth project, April 2020.

[16] M. Aldinucci, et al., “The Open Computing Cluster for Advanced data Manipulation (OCCAM),” in Journal of Physics: Conf. Series 898 (CHEP 2016), San Francisco, USA, 2017.

[17] Marco Aldinucci et al. HPC4AI, an AI-on-demand federated platform endeavour. ACM Computing Frontiers 2018, Ischia, Italy, 8-10 May 2018. doi: 10.1145/3203217.3205340

[18] StreamFlow: cross-breeding cloud with HPC Published in ArXiv 2020 https://arxiv.org/abs/2002.01558

[19] O. Sefraoui et all, "OpenStack: toward an open-source solution for cloud computing", Int. Journal of Computer Applications, vol 55, num3, 2012

[20] Xillinx examples, https://github.com/Xilinx/SDAccel_Examples