Chapter 11

The DeepHealth HPC Infrastructure:

Leveraging heterogenous HPC and cloud computing infrastructures for IA-based Medical solutions

Add ORCiD here (won't appear in final printed book)

Chapter Abstract

150 words—will not be printed in the final book, but used in the metadata for online discoverability.

Authors:

BSC: Eduardo Quiñones, Jesus Perales, Asaf Badouh, Santiago Marco

CEA: Fabrice Auzanneau, François Galea

TREE Technology: David González, José Ramón Hervás, Tatiana Silva

UNITO: Iacopo Colonnelli, Barbara Cantalupo, Marco Aldinucci, Enzo Tartaglione

UPV: Rafael Tornero, Jose Maria Martínez, David Rodriguez, Izan Catalán, Jorge García, Carles Hernández, José Flich, Roberto Paredes

1. Introduction

Deep learning (DL) is increasingly being considered to effectively support medical diagnosis of complex diseases. DL includes two main operations: (1) *training*, which refers to the process of generating a predictive model, in the form of a Deep Neural Network (DNN), based on large data-sets composed of biomedical information (e.g., medical images); and (2) *inference*, which refers to the process of predicting a diagnosis based on a reduced data-set.

Deep learning training is the most computationally intensive operation, requiring very large memory and computing power. The training operation is an iterative process, requiring to iterate many times over all the data-set samples to properly adjust the model, i.e., the weights of the DNN. In general, larger datasets allows to obtain predictive models with higher accuracy, which in turn, increases the running time needed for the training procedure.

This makes unfeasible to run training operations on general purpose computing systems, such as those available in hospitals, for large enough medical datasets, even when such computers are equipped with powerful processors featuring many-cores or GPU acceleration devices.

The exploitation of **the parallel capabilities of HPC infrastructures**, composed of dozens or hundreds of computing nodes, allows split the training operation into smaller datasets upon which parallel training operations can be applied. These methods are mandatory when the samples do not fit into a single computing node. As an example, the data-set provided by the FISABIO Foundation (a partner of the DeepHealth project) composed of Covid-19 images is 141 GB which does not fit into the memory of a single computing node (e.g., the Marenostrum 4 Supercomputer, features 96 GB of memory per node).

Moreover, training operations include data augmentation processes to transform original images to mitigate the problem of overfitting, that are applied on-the-fly, because each unique image from the dataset may require to be transformed in a different way at every iteration, i.e., epoch. Applying a sequence of image transformations on-the-fly is also computationally expensive. It is important to remark that medical imaging datasets are typically composed of extremely large images, further increasing the computing requirements.

DeepHealth has developed an HPC toolkit capable of efficiently exploit the computing capabilities of HPC infrastructures to execute DL training operations, in a **fully transparent way**. To do so, the data/computer scientists only need to describe in a file (CSV or JSON or XML or similar) the set of computing nodes to be used during the training process and launch it.

HPC facilities are not well suited for every kind of application. Queue-based workload managers that

PREPRINT

commonly orchestrate HPC centers cannot satisfy the strict time-to-solution requirements of the inference phase, and air-gapped worker nodes are not able to expose user-friendly web interfaces for data visualisation. To address these issues, the DeepHealth HPC toolkit also supports cloud environments, with innovative hybrid solutions to support private and public clouds and a new Workflow Management System (WMS) capable of scheduling and coordinating different steps of a DL pipeline on hybrid HPC/Cloud architectures.

Overall, the DeepHealth HPC toolkit offers to the computer vision and deep learning functionalities included into the European Computer Vision Library (ECVL) and the European Distributed Deep Learning Library (EDDL)[1], the HPC capabilities needed to efficiently exploit the computing capabilities of HPC and Cloud infrastructures. The two libraries are developed within the context of the DeepHealth project as well.

2. The Parallel Execution of EDDL Operations

EDDL is a general-purpose deep learning library initially developed to cover deep learning needs in healthcare use cases within the DeepHealth project. EDDL provides hardware-agnostic tensor operations to facilitate the development of hardware-accelerated deep learning functionalities and the implementation of the necessary tensor operators, activation functions, regularization functions, optimization methods, as well as all layer types (dense, convolutional and recurrent) to implement state-of-the-art neural network topologies.

In order to be compatible with existing developments and other deep learning toolkits, the EDDL uses ONNX[2] , the standard format for neural network interchange, to import and export neural networks including both weights and topology. As part of it design to run on distributed environments, the EDDL includes specific functions to simplify the distribution of batches when training and inference processes are run on distributed computing infrastructures. The EDDL serializes networks using ONNX to transfer weights and gradients between the master node and worker nodes. The serialization includes the network topology, the weights and the bias. To facilitate distributed learning, the serialization functions implemented in the EDDL allow to select whether to include weights or gradients.

Next sections present the parallel strategies implemented to execute the EDDL operations and so leverage HPC and cloud architectures.

2.1. COMPSs

COMPSs is a portable programming environment based on a *task model*, whose main objective is to facilitate the parallelization of sequential source code written in Java or Python programming languages, in a distributed and heterogeneous computing environment. In COMPSs, the programmer is responsible of identifying the units of parallelism (named *COMPSs tasks*) and the synchronization data dependencies existing among them by annotating the sequential source code (using annotations in case of Java or standard decorators in case of Python).

PREPRINT

[1] https://github.com/deephealthproject

[2] "Open Neural Network Exchange. The open standard for machine learning interoperability," [Online]. Available: https://onnx.ai/. [Accessed 31 October 2020].

Figure 1 shows a snipped (simplified for readability purposes) of the parallelisation of the EDDLL training operation with COMPSs. COMPSs tasks are identified with a standard Python decorator **`@task`** (lines 1 and 5). The **`IN`**, **`OUT`** and **`INOUT`** arguments define the data directionality of function parameters. By default, parameters are **`IN`**, and so there is no need to explicitly specify **`IN`** parameters. Moreover, when a task is marked with **`is_replicated=True`**, the COMPSs task is executed in all the available computing nodes for initialization purposes; otherwise, it executes on the available computing resources.

```
1.  @task (is_replicated = True)
2.  def build (model):
3.   # The model is created at each worker
4.   […]
5.  @task(INOUT = weights)
6.  def train_batch(model, dataset):
7.   # Executed at each worker
8.   […]
9.  def main():
10.  # A new model is created
11.  net = eddl.model([…])
12.  build(net)
13.  for i in range(num_epochs):
14.    for j in range(num_batches):
15.     weight[j] = train_batch(net,dataset)
16.    # Synchronize all weights from workers
17.    compss_wait_on(weight)
18.    # Update weights on the model
19.    update_gradients(net,weight)
```

Figure 1. A (simplified) snipped of EDDLL training operation parallelised with COMPSs.

The train iterates over **`num_epochs`** epochs (line 13). At every epoch, **`num_batches`** batches are executed (line 14), each instantiating a new COMPSs task (line 15) with an EDDLL train batch operation. All COMPSs tasks are synchronize at line 17 with **`compss_wait_on`**, and the partial weights are collected. The gradients of the model are then updated with the partial weights at line 19.

The task-based programming model of COMPSs is then supported by its runtime system, which manages several aspects of the application execution and keeps the underlying infrastructure transparent to the programmer. The COMPSs runtime is organised as a master-worker structure:

- The *master*, executed in the computing resource where the application is launched, is responsible for steering the distribution of the application and data management.
- The *worker(s)*, co-located with the Master or in remote computing resources, are in charge of responding to task execution requests coming from the Master.

One key aspect is that the master maintains the internal representation of a COMPSs application as a Direct Acyclic Graph (DAG) to express the parallelism. Each node corresponds to a COMPSs task and edges represent data dependencies (and so potential data transfers). As an example, presents the DAG representation of the EDDLL training operation presented in Figure 1.

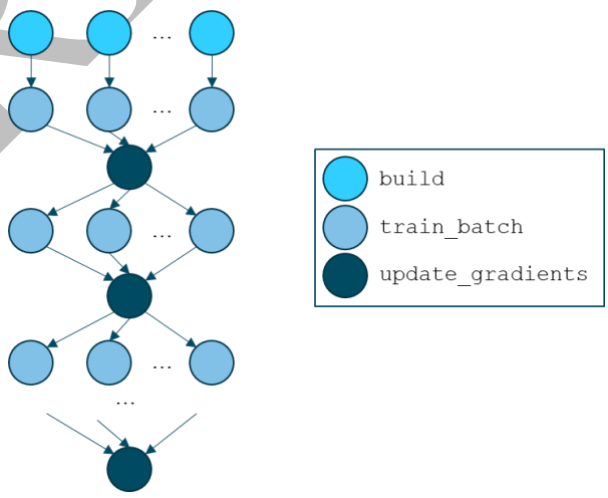

Figure 2. DAG representation of the application presented in Figure 1.

Based on this DAG, the runtime can automatically detect data dependencies between COMPSs tasks: as soon as a task becomes ready (i.e., when all its data dependencies are honoured), the master is in

PREPRINT

charge of distributing it among the available workers, transferring the input parameters before starting the execution. When the COMPSs task is completed, the result is either transferred to the worker in which the destination COMPSs tasks executes (as indicated in the DAG), or transferred to the master if a **`compss_wait_on`** call is invoked.

One of the main features of the COMPSs framework is that it abstracts the parallel execution model from the underlying distributed infrastructure. Hence, COMPSs programs do not include any detail that would tie them to a particular platform, boosting portability among diverse infrastructures and so enabling its execution in both a classical HPC environment and a cloud-based environment. To do so, COMPSs abstracts the underlying infrastructure by creating a set of execution environments, named COMPSs workers, in which COMPSs tasks execute. Internally, the COMPSs runtime implements different adapters to support the execution of COMPSs tasks in a given resource. Through a set of configuration files, the user specifies the available computing resources, which may reside in a computing cluster or in the cloud.

Figure 3 shows an example of this deployment. The execution starts in the *Computing Resource 1*, where the COMPSs Master executes. Then four workers are deployed in four different resources to distribute the workload, where the EDDLL training operations can be distributed.

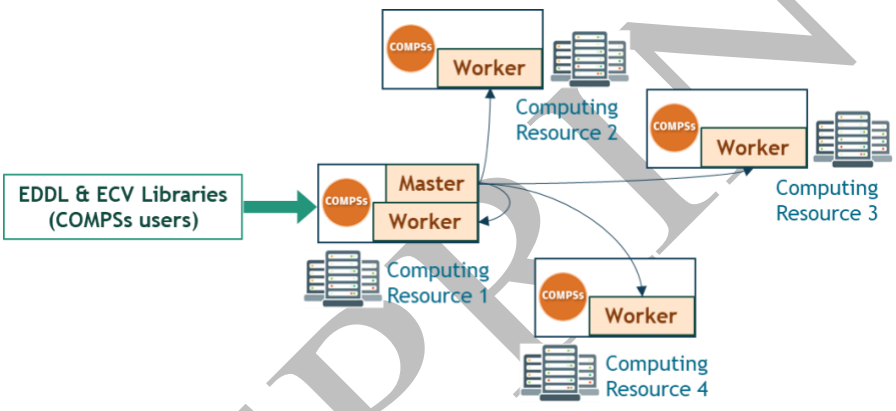

Figure 3. COMPSs deployment in a HPC infrastructure.

The COMPSs runtime is already supported in the Marenostrum Supercomputer[3] as a loadable module, in which the COMPSs workers are executed in the different Marenostrum computing nodes, each equipped with 2 Intel Xeon Platinum 8160 CPU with 24 cores each at 2.10GHz, 96 GB of main memory and 200 GB local SSD available as temporary storage during jobs. The COMPSs runtime is then responsible for distributing the parallelversion of the EDDLL training operation as described above.

2.2. StreamFlow

StreamFlow[4] is a novel Workflow Management System (WMS) capable of scheduling and coordinating different DL workflow steps on top of a diverse set of execution environments, ranging from practitioners' desktop machines to entire HPC centres. In particular, each step of a complex pipeline can be scheduled on the most efficient infrastructure, with the underlying run-time layer automatically taking care of worker nodes' lifecycle, data transfers, and fault-tolerance aspects.

The basic idea behind the StreamFlow paradigm is to easily express correspondences between the description of a coarse-grain application workflow, i.e., a graph containing the application steps with the related data dependencies, and the description of an execution environment, i.e., a manifest defining the capabilities of a target infrastructure. Starting from such description, the StreamFlow run-time layer is then able to orchestrate both the worker nodes' lifecycle and the execution of the application on top of them.

---

[3] https://www.bsc.es/marenostrum/marenostrum

[1] I. Colonnelli, B. Cantalupo, I. Merelli, and M. Aldinucci, "Streamflow: cross-breeding cloud with HPC," IEEE Transactions on Emerging Topics in Computing, 2020. doi:10.1109/TETC.2020.3019202

PREPRINT

With respect to the majority of WMSs on the market, StreamFlow gets rid of two common design constraints:

- There is *no need for a single shared data space* accessible from all the workers involved in a workflow execution. This allows supporting complex and hybrid execution infrastructures, including hybrid HPC/cloud architectures.
- Steps can be offloaded to *multi-agent execution environments*, ensuring the co-allocation of multiple and potentially heterogeneous worker nodes. This allows to offload steps involving distributed architectures, e.g., a multi-node DNN training driven by a COMPSs' master-worker infrastructure.

To provide enough flexibility, StreamFlow adopts a three-layered hierarchical representation of execution environments. A complex, multi-agent environment is called *model* and constitutes the unit of deployment, i.e., all its components are always co-allocated when executing a step. Each agent in a model, called *service*, constitutes the unit of binding, i.e., each step of a workflow can be bound to a single service for execution. Finally, a *resource* is a single instance of a potentially replicated service and constitutes the unit of scheduling, i.e., each workflow step is offloaded to a configurable number of resources to be processed. As an example, a Helm chart describing a COMPSs master pod and four COMPSs worker pods constitutes a model with two services, the former with one resource and the latter with four resources.

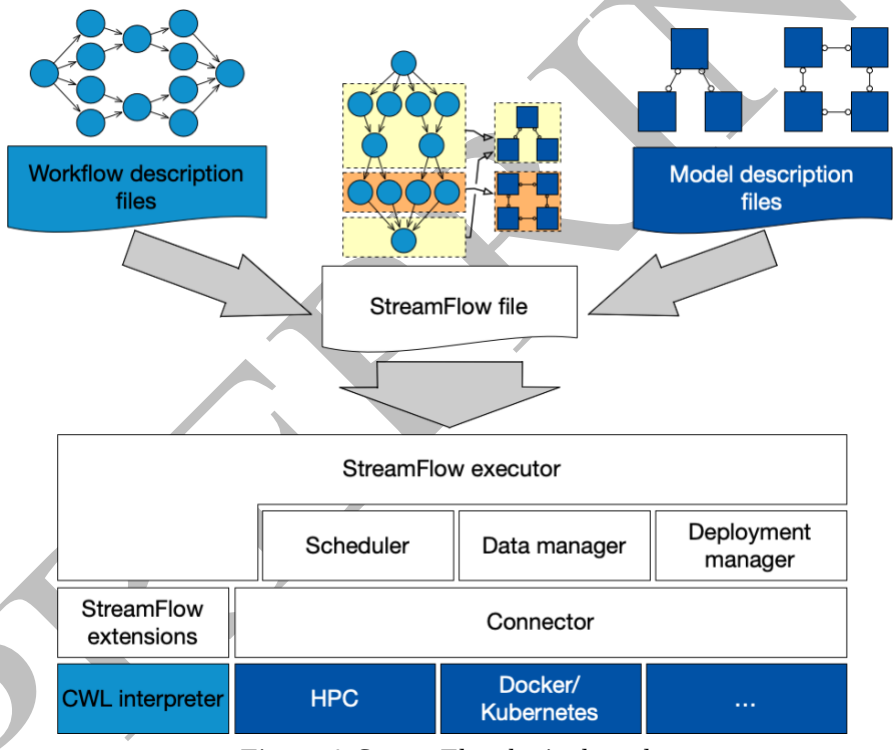

Figure 4. StreamFlow logical stack.

Figure 4 shows the StreamFlow's logical stack. The Deployment Manager is the component in charge of creating and destroying models when needed. To do that, it mainly relies on external orchestration technologies (e.g., Slurm, Kubernetes or Docker Compose) by means of pluggable implementations of the Connector interface. After a model is deployed, the Scheduler component is in charge of selecting the best resource on which each workflow step should be executed while guaranteeing that all the requirements are satisfied. Finally, the Data Manager, which knows where each step's input and output data reside, must ensure that each service has access to all the data dependencies required to complete the assigned workflow step, performing data transfers only when necessary.

Instead of coming with yet another way to describe workflow models, StreamFlow relies on an existing coordination format, called Common Workflow Language (CWL). CWL is an open standard for describing analysis workflows, following a declarative JSON or YAML syntax. Being CWL a fully

declarative language, it is far simpler to understand for domain experts than its Make-like or dataflow-oriented alternatives. Moreover, the fact that many products offer support for CWL, either alongside a proprietary coordination language or at higher-level semantics on top of low-level APIs, fosters portability and interoperability.

It is also worth noting that StreamFlow does not need any specific package or library to be installed on the target execution architecture other than the software dependencies required by the host application (i.e., the involved workflow step). This agentless nature allows virtually any target architecture reachable by a practitioner a potential target model for StreamFlow executions, as long as a compatible Connector implementation is available.

3. Cloud Infrastructures

3.1. Hybrid Cloud

The hybrid cloud is the combination of Private Cloud and Public Cloud, allowing the exploitation of the best of both types. There are several reasons why it may be interesting to have a hybrid cloud. The three most typical cases are the following:

- **Security**: Sensitive information that cannot be at risk, e.g., banking or health information, is kept into the private part of the cloud to minimize potential risks.
- **Cost**: The presence of sporadic peaks on the computing workload can be served with public cloud resources, without requiring dimensioning the private cloud infrastructure to compensate the peaks. By doing so, the hybrid cloud part is used for the usual load, and the public part is used to compensate the peaks while keeping the service alive.
- **Availability**: In the same way as with a Content Delivery Network (CDN), a hybrid private – multi-public cloud can be created. Multi-public clouds can be chosen to be close to customers to reduce latency and maximize available bandwidth, to reduce cost or to improve availability.

The *DeepHealth HPC toolkit* includes an on-premise private cloud based on a Kubernetes cluster (and hosted by the DeepHealth partner TREE Technology), and a public cloud in AWS computing resources, as illustrated in Figure 5.

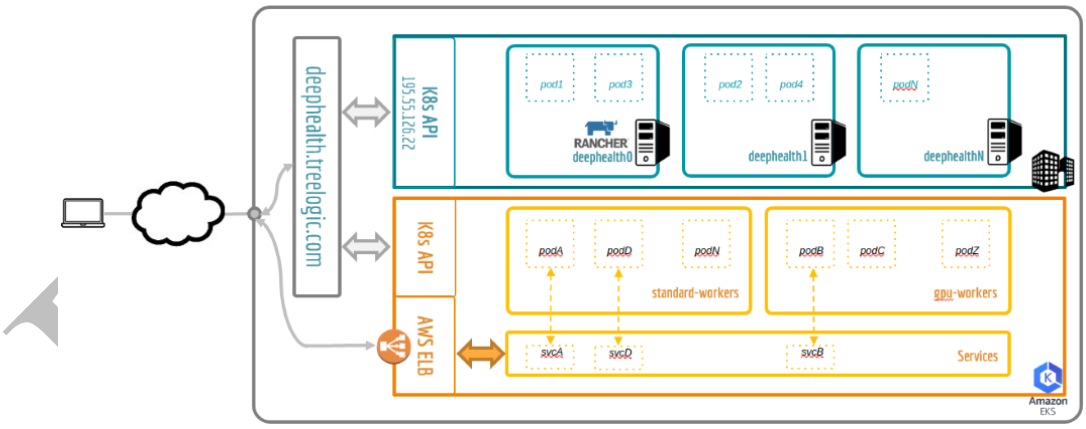

Figure 5. Hybrid Cloud Environment.

By doing so, the DeepHealth hybrid cloud allows to provision computing resources in a flexible way to accommodate project requests. GPU computing capabilities can be added – both through provisioning on the on-premise Kubernetes cluster and via nodes provisioned using Amazon Elastic Kubernetes Service (EKS) technology on Amazon Web Services (AWS). This hybrid infrastructure has two main pieces: Rancher and API Services.

1. The open-source Rancher multi-cluster orchestration platform is used[5] to support the management of the clusters. Rancher allows managing the operational and security challenges on multiple

[5] https://rancher.com/docs/rancher/v2.x/en/

Kubernetes clusters across any infrastructure. Moreover, its web interface provides control over deployments, jobs, pipelines, including an app catalog for fast deployment using the Helm software package manager[6].

2. An API has been developed to facilitate the use and/or integration of the libraries with the Kubernetes platform. Moreover, a high-level REST API has been developed on top of the Kubernetes API to help abstract the user from the infrastructure itself, simplifying the deployment and management of the workflows. It provides functions of varying complexity, from simple ones, like list of Pods, to more complex ones such as expose Pods – which implements functionality abstracting the user from the potentially complex configuration of the clusters (e.g., multi-cloud, hybrid cloud, etc.). The API itself can support the addition of new Kubernetes clusters both on-premise and in the cloud from any provider. To guarantee properly authenticated and authorized access to the DeepHealth cloud, a connection through a VPN is required to access the API.

3.2. Parallel Execution on Cloud Environments

DeepHealth exploits the parallelisation of DL operations on cloud enviroments by the usage of the COMPSs and StreamFlow parallel frameworks presented in Section 2. Next sections describe them.

3.2.1. Parallel Cloud Execution based on COMPSs

Unlike the Linux-based infrastructure, there is no need for setting up the execution environment in all the computing resources, but only a docker image must be available, e.g., Docker Hub[7]. Figure 6 shows an example of this deployment. The execution starts in the *computing resource 1*, where the COMPSs Master executes. Then three workers are deployed in three different containers in the cloud infrastructure (in our case the cloud is provided by TREE), where the COMPSs application is distributed (in our case, EDDLL training operations).

Moreover, the COMPSs runtime is being adapted to support the cloud infrastructure provided by TREE. The cloud is based on Kubernetes (K8S)[8], and allows to manage applications in a container technology environment and to automate the manual processes to deploy and scale containerized applications. Moreover, an API is being developed by TREE to help abstracting the user from the infrastructure itself, speeding up the processes of deployment and management of the workflows. COMPSs runtime interacts with this API to deploy workers and distribute the workload.

The DeepHealth hybrid cloud has been integrated with the COMPSs framework (see Section 2.1), which allows to accelerate the Deep Learning training operations by dividing the training data sets across a large set of computing nodes available on cloud infrastructures, and upon which partial training operations are then be performed. This combination is done through the REST API, allowing COMPSs to abstract from the hybrid infrastructure and perform their workloads regardless of where they are deployed.

Figure 6 shows a possible distribution of the execution of the parallel version of the EDDLL training operation presented in Section 2.1, in the DeepHealth hybrid cloud considering three COMPSs workers, setting the number of replicas to 3. The COMPSs runtime use the DeepHealth cloud API to automatically deploy the master and the three replicas in which the COMPSs workers will be executed. Once the deployment is completed, the parallel execution of the training operation is initiated, and so the COMPSs runtime starts the distribution of the different COMPSs tasks (i.e., the **`build`** and **`train_batch`** tasks shown in Figure 2, guaranteeing the data dependencies among tasks. In this case, the **`update_gradients`** function is executed in the COMPSs master to aggregate the partial computed weights at the end of each epoch.

PREPRINT

[6] https://helm.sh/
[7] https://www.docker.com/products/docker-hub
[8] https://kubernetes.io/

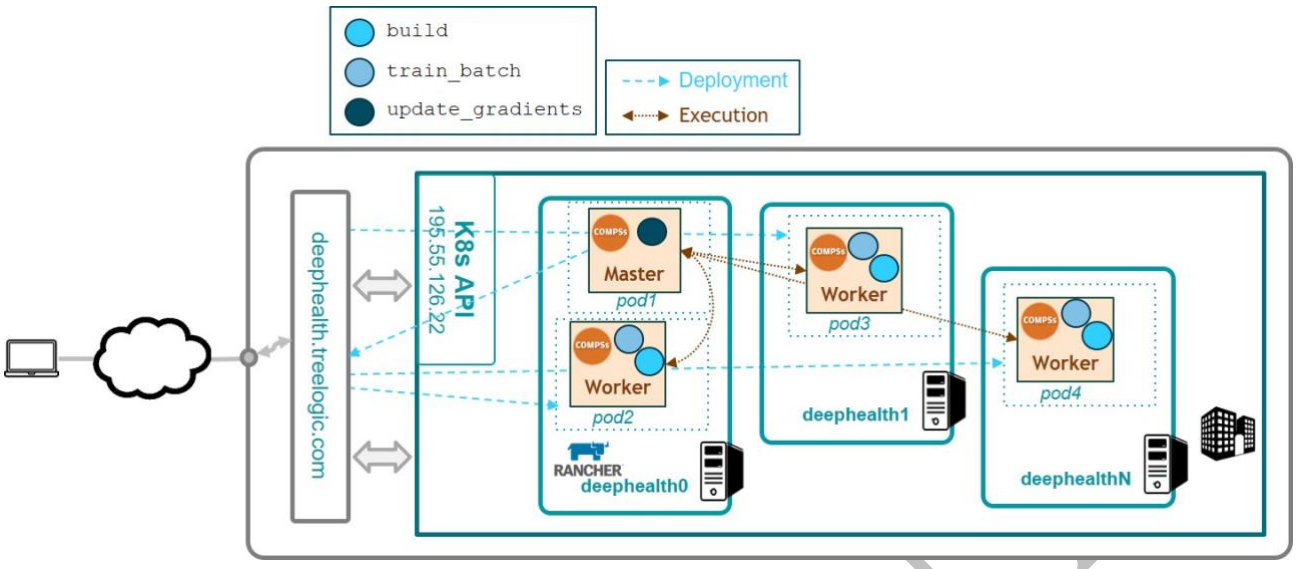

Figure 6. Distributed EDDLL training on the DeepHealth Hybrid Cloud.

3.2.2. Parallel Cloud Execution based on Stream-Flow

StreamFlow relies on the University of Torino's OpenDeepHealth (ODH), which implements a hybrid HPC/cloud infrastructure to effectively support the training and inference of AI models. The WMS introduced in Section 2.2, is the key technology enabling a transparent offloading of AI tasks to heterogeneous sets of worker nodes, hiding all deployment and communication complexities to the expert users.

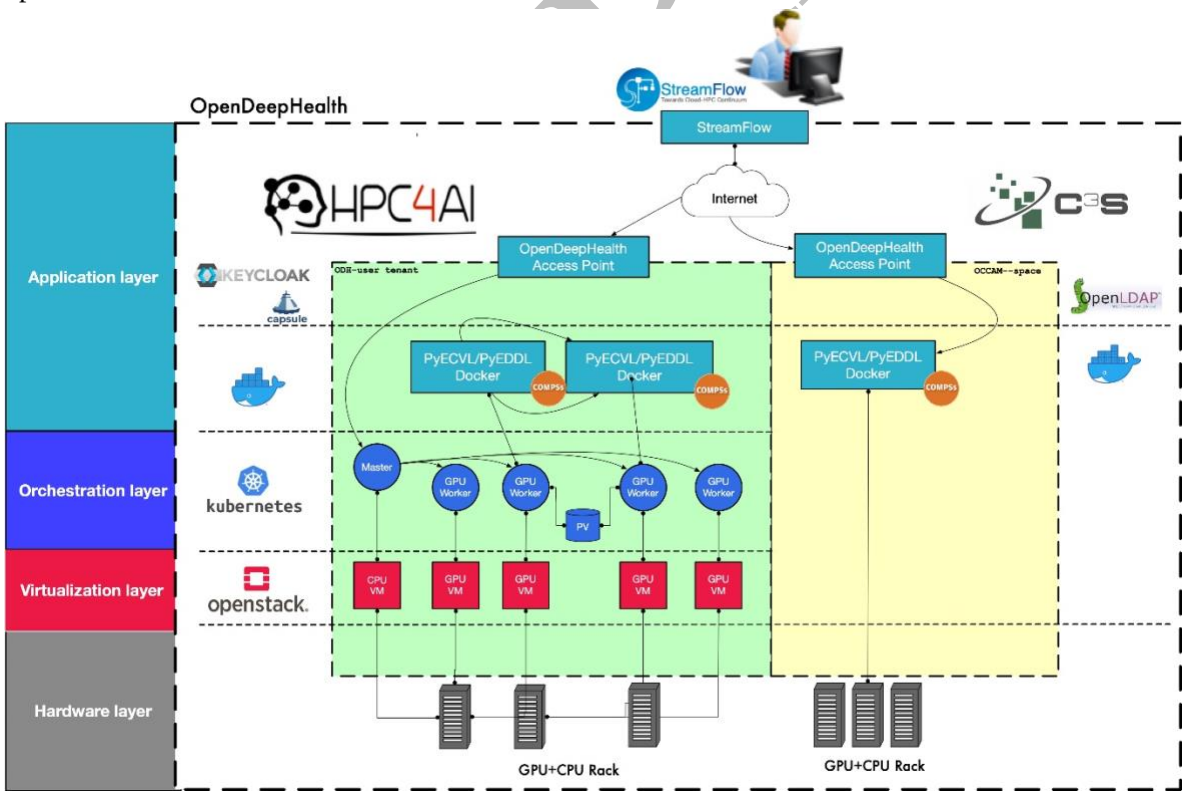

Figure 7. OpenDeepHealth Architecture.

As shown in Figure 7, ODH includes two different infrastructures:

- The HPC component is the C3S OC--CAM (Open Computing Cluster for Advanced data

Manipulation)[9] cluster, with 46 heterogeneous nodes also including NVIDIA Tesla K40 and V100 GPUs. Workloads are orchestrated through an elastic virtual farm of hardened Docker containers, running directly on top of the bare metal layer.

- The cloud component is the HPC4AI[10] private cloud, a multi-tenant Kubernetes cluster running on top of an OpenStack cloud. The underlying physical layer consists of high-end computing nodes equipped with Intel Xeon Gold 80-cores and 4 NVIDIA Tesla T4 GPUs per node.

ODH implements a novel form of multi-tenancy called *"HPC Secure multi-Tenancy"* (HST), specifically designed to support AI application on critical data. HST allows resource sharing on a hybrid infrastructure while guaranteeing data segregation among different tenants, both inside the cloud and between the HPC and cloud components. Moreover, access to the tenant is mediated by an identity manager, guaranteeing identity propagation across the entire environment.

ODH platform fully integrates the DeepHealth toolkit via Docker containers, both on bare metal and in the multitenant Kubernetes cluster. The DeepHealth toolkit provides functionalities to be used for both training and inference, addressing the complexity of the different available computational resources and target architectures at both the training and inference stages. The training phase is performed by AI experts mainly in research-focused environments using specialized architectures like HPC centres with FPGAs and GPUs because the main goal is optimizing the number of samples processed per second without hindering the overall accuracy. In the inference step, based on pre-trained models and deployed in production environments (and even small devices in the edge), response time for prediction of a single sample is crucial.

Leveraging on the StreamFlow capability to coordinate tasks running on different execution environment, each step of an AI pipeline can be executed on the computational component that is more appropriate according to the specific characteristic of the computation. For instance, the computational-heavy training step can be initially tested on a multiGPUs node and then executed on the OCCAM HPC cluster, while the much more lightweight inference step, can be offloaded to a CPU-equipped Kubernetes worker node in HPC4AI allowing, when needed, an interactive inspection of the final results. An AI expert can simply launch the pipeline directly from his computer using StreamFlow, which orchestrates the execution of the first step on OCCAM and the second one on HPC4AI. It also manages all the required data transfers in a fully transparent way. We demonstrated this approach with the Lung Nodule Segmentation AI pipeline, a DeepHealth project use case (see Figure 8) that will be described in Chapter 12.

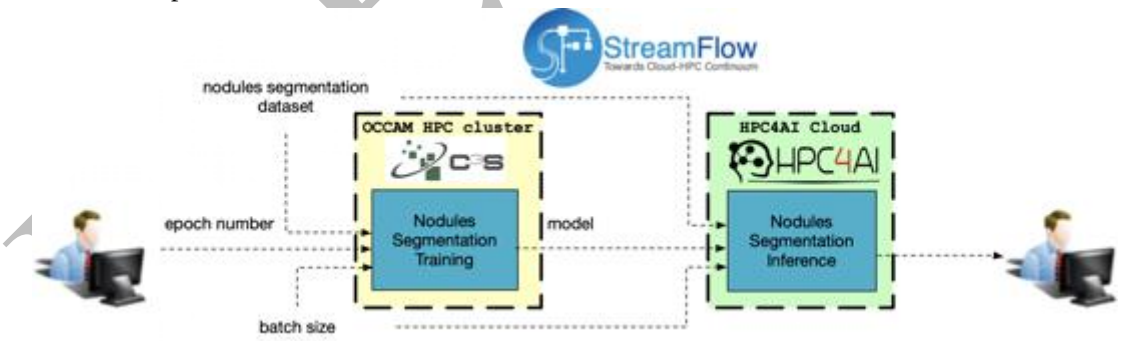

Figure 8. A DeepHealth Use Case AI Pipeline executed using StreamFlow.

Moreover, integration between StreamFlow and COMPSs provides the ability to perform the distributed training in the ODH cloud environment, by using the clouds capabilities of COMPSs

The idea behind this overall approach is that the ability to deal with hybrid workflows can be a crucial

[9] M. Aldinucci and others, "OCCAM: a flexible, multi-purpose and extendable HPC cluster," *Journal of Physics: Conference Series*, vol. 898, 2017

[10] M. Aldinucci, S. Rabellino, et al. "HPC4AI, an AI-on-demand federated platform endeavour," in *Acm computing frontiers*, Ischia, Italy, 2018. doi:10.1145/3203217.3205340

DRAFT

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.

4. Acceleration Devices: GPU and FPGAs

4.1. FPGA Acceleration

FPGAs are devices with reconfigurable logic that can be combined and interconnected in order to process a specific algorithm. Contrary to CPUs and GPUs, in an FPGA the architecture (design) is adapted to the algorithm, thus, offering an opportunity to their optimization. On the other hand, the resources of an FPGA device are limited and usually the device works at a lower frequency. Therefore, a careful and custom design is needed in order to take benefits out of it. Although FPGAs are well suited for inference in Deep Learning, they can also be used in specific situations for training processes.

FPGAs are supported in the DeepHealth project in two orthogonal although complementary and needed directions. First, large FPGA infrastructures are being adapted to the project by suitable interfaces and protocols specific of FPGAs. Second, specific and optimized FPGA algorithms are developed for specific use cases within the project. In the next sections we describe the adaptations and developments performed for the infrastructure and then the optimizations at algorithmic level.

4.1.1. The DeepHealth FPGA infrastructure

The Mango FPGA Platform

The DeepHealth toolkit includes the MANGO FPGA platform, developed within the European MANGO project. As part of DeepHealth activities we are evolving this cluster of FPGAs from a hardware prototyping platform to a high performance and low energy compute platform. The platform consists of two clearly differentiated subsystems: the General-purpose Nodes (GNs) and the Heterogeneous Nodes (HNs). The former executes the host applications, as well as the low-level communication libraries, run; the latter represent the computational part of the system built upon the FGPA modules:

- A GN (see Figure 9) consists of a Supermicro SuperBlade module SBI-7128RG-F equipped with an Intel Xeon processor E5-2600 v3, 64 GB of RAM memory and 1 TB of SSD storage. Each GNs is connected via PCIe to two HNs, so it can use both HN subsystems. GNs are also connected to the HNs via Ethernet and USB for cluster programming and management purposes.
- An HN (see Figure 9 and Figure 10) consists of 12 FPGA modules mounted on top of 4 proFPGA motherboards and placed in an FPGA cluster. This setup is extended with a total amount of 22 GB of RAM memory split in several DDR3 and DDR4 modules among different FPGAs. The HN is a heterogeneous subsystem since it contains different types of FPGAs, it is composed by Xilinx Kintex Ultrascacle XCKU-115, Xilinx Virtex 7 V2000T, Xilinx Zynq 7000 SoC Z100 and Intel Stratix 10 SG280H FPGA modules. The HN cluster is also equipped with FPGA interconnection cables to enable direct communications between its FPGAs. The cables are disposed in such a way that maximize communication bandwidth among the overall system but keeping the throughput balance between the different FPGA modules. The HN also includes one PCIe extension board that enables PCIe communications with the GN. Figure 10 shows the positioning and interconnections of the different FPGA / memories and cables between the different modules.

Overall, the complete DeepHealth FPGA-based infrastructure consists of a total of 4 GNs and 8 HNs arranged in two cabinets as shown in Figure 9.

PREPRINT

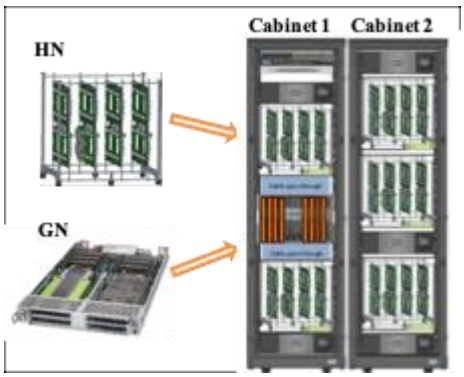

Figure 9. MANGO Hardware: GNs and HNs.

Figure 10 HN modules and interconnection.

With the objective of facilitating programmability, the DeepHealth FPGA platform the FPGA in two partitions: a static partition, known as *shell*, and a reconfigurable partition (see Figure 11). The former remains unchanged while the host is up and running. When the host is booting this partition is loaded into the FPGA from an external memory drive. Once uploaded, it provides the required interfaces to communicate with attached peripherals and the host where the FPGA is connected to in an efficient way. In the MANGO cluster the connection to the host is accomplished through a PCIe Gen3 x8 bus, offering a bidirectional raw bandwidth of 16GB/s, approximately. In addition, the static partition provides the clocks and reset networks to the rest of the elements in the FPGA, such as kernels. On contrary, the reconfigurable partition consists of a placeholder, inside of the FPGA, that can be changed at runtime using the features that the *shell* provides. This partition contains the resources in which application kernels can be deployed.

The MANGO cluster is compatible with the OpenCL application programming interface (API)[11] for FPGA initialization, data transfer and kernel offloading, supporting both Xilinx and Intel FPGAs. Kernels are the part of the application that will run on the FPGA, so as to provide acceleration capabilities to the applications.

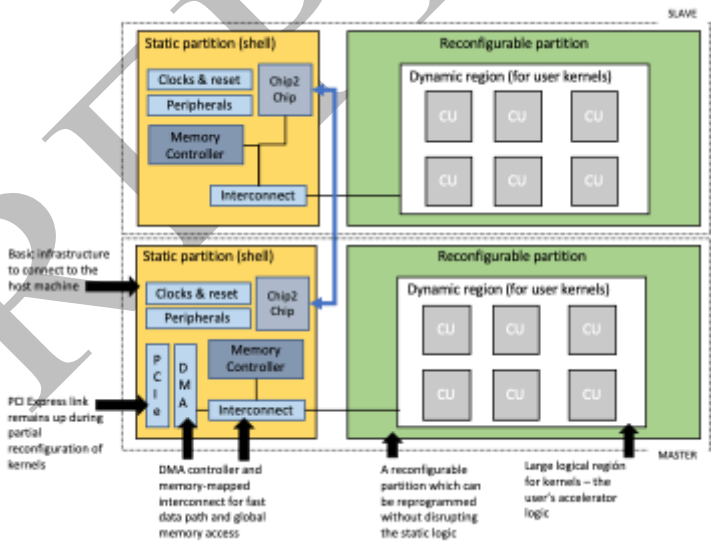

Figure 11. FPGA logical design of two FPGAs in the same cluster connected with Chip2Chip.

Furthermore, the FPGA design has been instrumented to support communication between the different FPGAs an HN is composed of. On the one hand, FPGA-to-FPGA connections within a cluster use the I/O pins available on the devices. On the other hand, connections between FPGAs at different clusters use the so-called Multi-Gigabit Transceivers (MGT). With these communication channels the GNs has DMA access to all the devices in an HN cluster, as well as to offload and control application kernels running on different FPGAs. At the same time, HN clusters are able to communicate with each other.

[11] Khronos.org/opencl

PREPRINT

To accomplish these goals the design incorporates an IP core provided by Xilinx named Chip2Chip(C2C). This IP core works like a medium access bridge connecting two devices over a memory-mapped interface in compliance with AXI protocol specifications.

C2C can be configured to work in master or slave mode, as depicted inFigure 11. To connect two FPGAs with C2C one device has to be configured as master and the other device as slave. The rest of the FPGAs in a cluster can be connected similarly, following a daisy chain. In this architecture, the host has access to the memory on the slave FPGA through the C2C instances. Equally, provided the required logic is available, a kernel executing on the master FPGA can access the memory of the slave FPGA. As a result, this communication interface enables the host to offload and control kernels not only in the master FPGA, but also in the slave one.

Multi-FPGA support

Another feature being researcher within the DeepHealth FPGA infrastructure is the analysis on the use of multiple FPGAs for the implementation for a single inference network. Uniting the compute and memory resources of all used FPGAs makes possible to implement neural networks with higher requirements of memory (weights) and/or with a higher level of parallelism, reducing the latency.

The N2D2 (Neural Network Design & Deployment) deep learning framework made by DeepHealth partner CEA[12] (see Section 4.1.3 for further details), features a technology called dNeuro[13], which is able to export an inference network as a hardware description in RTL language, suitable for synthesis and implementation on a FPGA. dNeuro is optimized to use the available compute resources (mainly, DSPs) as efficiently as possible, and to use only the embedded memory resources (block RAMs), also in the scope of efficiency. dNeuro generates the network according to specified constraints, allowing more or less parallelism depending on the number of available DSP units. Specifying a number of DSPs higher than in a single FPGA results in a network suitable for multi-FPGA execution.

CEA is developing technology to map netlists automatically onto a multi-FPGA platform, similarly as if this platform were a single, larger FPGA. For this, the netlist has to be split into multiple netlists, one for each FPGA. The partitioning must be as efficient as possible, in order to maintain efficient communication between the FPGAs, both in terms of critical path preservation and control of the number of inter-FPGA signals. To achieve this goal, specifically adapted state-of-the-art hypergraph partitioning techniques are being used to ensure the overall performance improvement of the application.

4.1.2. An Optimised FPGA Board Design for DL

DeepHealth is also investigating on the development of a FPGA board (see Figure 12) optimized for inference. The key component of the board is an INTEL Stratix-10 MX1650 or MX2100 FPGA. Both types are package and pin compatible and have FPGA internal High Bandwidth Memory (HBM) embedded. 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 board designs.

PREPRINT

[12] https://github.com/CEA-LIST/N2D2
[13] https://www.cea.fr/cea-tech/leti/Documents/démonstrateurs/Flyer_DNEURO.pdf

Figure 12. The new FPGA board.

Furthermore, 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 depends 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.

For direct host communication a PCIe interface (Gen3 x 16) is implemented. To communicate with external devices via high-speed interfaces four QSFP28 interfaces are available at the bracket. Each of these interfaces provides a full-duplex bandwidth of 100GBit/s. To interconnect multiple of these FPGA boards or to further extend the capabilities for external communication general purpose connectors are available at the backside of the board.

Finally, a board support package (BSP) is provided to use OpenCL/HLS tools for programming and integration of the hardware into the DeepHealth infrastructure.

4.1.3. FPGA-based Algorithms

Pruned and quantized models enable the use of FPGA devices for energy-efficient inference processes when compared to GPUs or CPUs. This section presents the use of these two techniques in the DeepHealth FPGA infrastructure

Quantization and N2D2 and interface with DeepHealth via ONNX

N2D2 is a comprehensive solution for fast and accurate DNN simulation and full and automated DNN-based applications building. It integrates database construction, data pre-processing, network building, benchmarking and hardware export to various targets. It is particularly useful for DNN design and exploration, allowing simple and fast prototyping of DNN with different topologies.

Once the training DNN performances are satisfying, an optimized version of the network can be automatically exported for various embedded targets. An automated network computation performances benchmarking can also be performed among different hardware targets. Various targets are currently supported by the tool-flow: from plain C code to C code tailored for High-Level Synthesis (HLS) with Xilinx Vivado HLS and code optimized for GPU. Various optimizations are possible in the exports: (1) DNN weights and signal data precision reduction (down to 8 bit integers); (2) non-linear network activation functions approximations; and (3) different weights discretization methods.

The post-training quantization algorithm is done in three steps:

1. *Weights normalization:* all weights are rescaled in the range [−1.0,1.0],
2. *Activations normalization:* activations at each layer are rescaled in the range [−1.0,1.0] for signed outputs and [0.0,1.0] for unsigned outputs. The optimal quantization threshold value of the activation output of each layer is determined using the dataset, and implies the use of

PREPRINT

additional shifting and clipping layers.

3. *Quantization:* inputs, weights, biases and activations are quantized to the desired precision.

Interworking between the N2D2 framework and the EDDL library is possible thanks to the ONNX file exchange format. This allows any Neural Network designer to get advantage of both environments. This integration is illustrated with the two flows outline hereafter.

In order to obtain an inference code making an intensive usage of integer operators rather than floating-point ones, the flow is the following:

1. Regular training performed in EDDL;
2. export by EDDL of the trained Neural Network to ONNX;
3. import of the ONNX network into N2D2;
4. 8-bit quantization of the network using N2D2;
5. export the quantized network to ONNX;
6. import the ONNX into EDDL;
7. generation of the inference code using EDDL facilities.

This flow is schematized on Figure 14. It can be adjusted to take advantage of the various automated inference code generators proposed by N2D2.

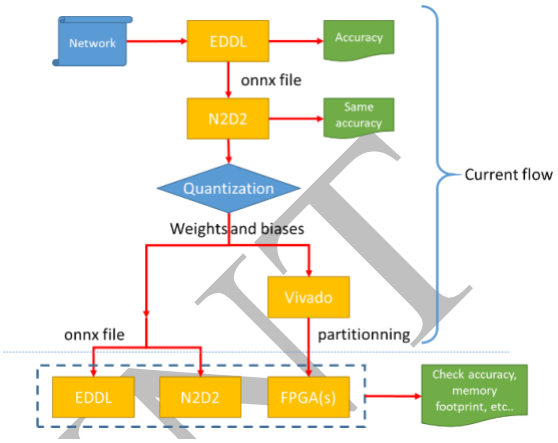

Figure 13. Quantization flow with EDDL

FPGA based HPC oriented flows are as follows:

1. N2D2 users to access to the FPGA based HPC inference code generation provided by the EDDLL library;
2. EDDL users to benefit for N2D2 training capabilities not available in EDDL yet;
3. regular training performed in N2D2;
4. 8-bit quantization of the network within N2D2;
5. export the quantized network to ONNX;
6. import the ONNX Neural Network into EDLL;
7. generation of the inference code using EDDLL facilities.

The performance of the ISIC classification use-case generated through this flow are detailed below.

With 63% classification accuracy, it shows up the same accuracy as the floating-point version with a memory footprint divided by four. The storage memory size for the weights and biases of the standard VGG16 network is estimated to 512.28 Mega bytes. When performing quantization, the weights are stored in signed 8 bits integers, and biases in signed 16 bits integers. Each layer requires an additional scaling factor, stored in an unsigned 8-bit integer. Coming from 32 bits values, the required storage size will be divided by almost 4. The storage memory size for the weights and biases of the quantized VGG16 network is estimated to 1 074 445 952 bits, which is 128.08 MB.

Similar results were obtained with different versions of the MobileNet network[14]. The original version of MobileNet was trained on the ISIC dataset: recognition rates ranged from 73% to 79% depending on the alpha factor (from 0.25 to 1) which drastically impacts the required memory size. Eight bits quantization and export times were in the range of 150 to 500 seconds, and the recognition rates of the quantized networks remained in the same range as before. The estimated memory footprint was 440 kB (alpha = 0.25), 1250 kB (0.5) and 4050 kB (1) with estimated frame rate on FPGA ranging from 250 to 1780 frames per second.

DNN Compression Methods

[14] https://arxiv.org/abs/1704.04861

PREPRINT

It is well known that many DNNs, trained on some tasks, are typically over-parametrized.[15] DeepHealth also includes pruning techniques. The goal of these techniques is to achieve the highest sparsity (i.e. the maximum percentage of removed parameters) with minimal (or no) performance loss. One possible approach relies in a regularization strategy, used at training time, employing the use of the *sensitivity* term[16] in order to penalize the parameters which are not useful towards the solution of the target classification task:

$$S_w = \frac{1}{C}\sum_{k=1}^{C}\left|\frac{\partial y_k}{\partial w}\right|$$

where C is the number of the output classes, y_k is the k-th output of the model and w is the evaluated parameter. The lowest S_w, the least its change will perturbate the change of the output. Through this metric, it is possible to remove parameters which impact the least the model's performance. It has been recently shown that iterative pruning strategies enable the achievement of higher compressions, justifying the relative training overhead towards one-shot approaches.[17]

Recently, a lot of attention has been devoted to the problem of the so-called structured pruning: unless focusing on single parameters, which enable a reduced gain in terms of FLOPs and memory footprint at training time, pruning algorithm should be focused on removing entire neurons. Comparing LOBSTER,[18] which is a state-of-the-art un-structured pruning algorithm which removes a very large quantity of parameters, to SeReNe,[19] which is a sensitivity-based structured pruning algorithm, results that structured pruning strategies, despite removing less parameters than the unstructured ones, bring significant advantages in terms of memory footprint and FLOPs reduction, achieving up to $2\times$ footprint saving and $2\times$ FLOPs reduction, evaluated at inference time on resource-constrained devices, for complex architectures like ResNet and VGG-16, trained on state-of-the-art tasks, with no performance loss.

Development of DL Kernel on the DeepHealth FPGA Infrastructure

DL kernel are specialized on the most frequent neural network layers used in the specific domain of the project (image-related classification and segmentation processes for health), therefore convolutions, resizing, and pooling operations are the focus of the kernel. However, the design of the kernel is performed using high-level synthesis (HLS). With HLS an algorithm described in high-level language such as C++ is transformed into an implementation on a reconfigurable device or even into an ASIC.

The kernel design follows the dataflow model by Xilinx combined with the use of streams. With this model a pipelined design between modules can be created and data can be processed concurrently on all the connected modules. The use of streams enables concurrency.

PREPRINT

[15] H. N. Mhaskar, T. Poggio, Deep vs. shallow networks: An approximation theory perspective, Analysis and Applications 14 (06) (2016) 829–848.
[16] E. Tartaglione, S. Lepsøy, A. Fiandrotti, G. Francini, Learning sparse neural networks via sensitivity-driven regularization, in: Advances in Neural Information Processing Systems, 2018, pp. 3878–3888.
[17] Tartaglione, Enzo, Andrea Bragagnolo, and Marco Grangetto. "Pruning artificial neural networks: a way to find well-generalizing, high-entropy sharp minima." *International Conference on Artificial Neural Networks*. Springer, Cham, 2020.
[18] Tartaglione, Enzo, et al. "LOss-Based SensiTivity rEgulaRization: towards deep sparse neural networks." *arXiv preprint arXiv:2011.09905* (2020).
[19] Tartaglione, Enzo, et al. "SeReNe: Sensitivity based Regularization of Neurons for Structured Sparsity in Neural Networks." *arXiv preprint arXiv:2102.03773* (2021).

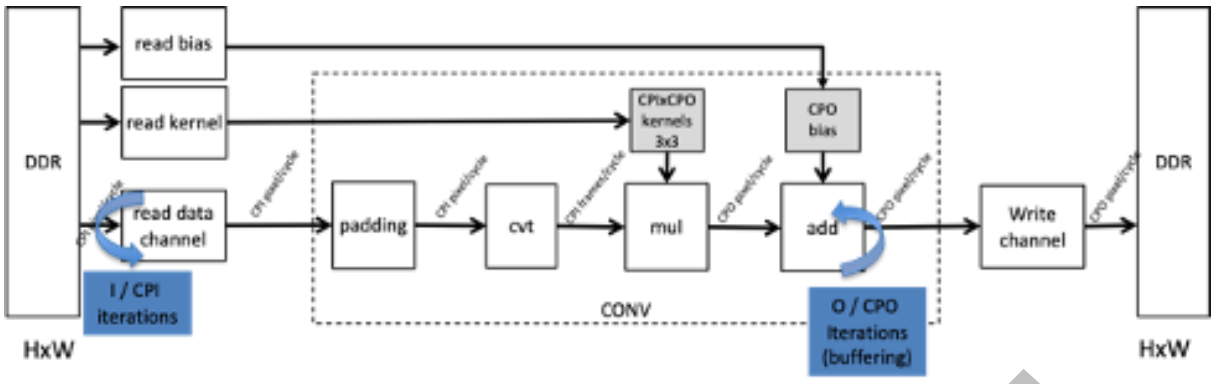

Figure 14. Baseline kernel design.

Figure 14 shows the baseline design for the kernel targeting convolutional operations. Each box represents a different module and arrows represent streams. The kernel reads data (activations, bias, and weights) from a DDR memory attached to the FPGA and produces features being written back to the DDR memory. The images (activations) are pipelined through all the modules. The padding module provides padding support to the input images read in a streamflow fashion and then forwards the padded image to the next module. The cvt module converts the input stream into frames of pixels that will be convolved in the mul module and reduced in the add module. Finally, the produced features are written back to memory.

The design supports parallel access to memory in order to processes a defined number of input (CPI) and output (CPO) channels. Therefore, the design can be customized to different granularities of the convolution operation. Indeed, the number of convolutions operations performed in each cycle is CPI x CPO as each CPI channel is used for each CPO channel (direct convolutions supported). Figure 15 shows the parallel multiplications performed and the frames generated by the cvt module.

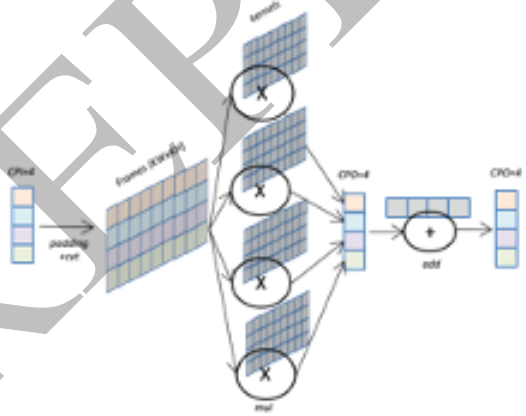

Figure 15. *Multiplications in 2D convolution kernel.*

4.2. Many-core and GPU Acceleration

The EDDL implementation is optimized to efficiently execute on a single computing node featuring many-core fabrics or a set of GPU cards accelerators. In that regard, the EDDL includes an API to build the neural network and the associated data structures (according to the network topology) on the following acceleration technologies (named *Computing Service* in EDDL nomenclature): CPU, GPU and FPGA. Tensor operations are then performed using the hardware devices specified by means of the Computing Service provided as a parameter to the build function. Moreover, the number of CPU cores, GPU cards or FPGA cards to be used are indicated by the Computing Service. This section presents the many-core and GPU acceleration included in the EDDL. See Section 4.1 for FPGA acceleration.

PREPRINT

On many-core CPUs, tensor operations are performed by using the Eigen[20] library that rely the parallelization on OpenMP[21]. When using GPU cards, the forward and backward algorithms are designed to minimize the number of memory transfers between the CPU and the GPU cards in use, according to the configuration of the Computing Service. EDDL incorporates three modes of memory management to address the lack of memory when a given batch size does not fit in the memory of a GPU. The most efficient one tries to allocate the whole batch in the GPU memory to reduce memory transfers at the minimum, the intermediate and least efficient modes allow to work with larger batch sizes at the cost of increasing the number of memory transfers to perform the forward and backward steps for a given batch of samples.

In the case of using more than one GPU in a single computer, the EDDL internally creates one replica of the network per GPU in use, and automatically splits every batch of samples into sub-batches, one sub-batch per GPU, so that each sub-batch is processed by one GPU. Every time a batch is processed, the weights stored in each GPU are different and weight synchronization every certain number of batches is required to avoid divergence. The weight synchronisation is done by transferring all the weights from GPU memory to CPU memory, computing the average, and transfer back the updated weights to the memory of all the GPU cards in use, i.e., to all the replicas of the network. The computing service used for defining the use of GPU cards has an attribute to indicate the number of batches between weight synchronizations. As memory transfers between CPU and GPU must be reduced as much as possible, a trade-off between performance and divergence must be reached by means of this attribute, whose optimal value will vary depending on the data set used for training and the topology of the neural network.

EDDL support for GPUs has been implemented twice, by means of CUDA kernels developed as part of the EDDL code, and by integrating the NVIDIA cuDNN library. The use of different hardware accelerators is completely transparent to developers and programmers who use the EDDL; they only need to create the corresponding Computing Service to use all or a subset of the computational resources.

5. Conclusions

DeepHealth HPC toolkit allows the European Distributed Deep Learning Library (EDDL) to be efficiently executed on HPC and Cloud infrastructures. On one side, it includes HPC and cloud workflow managers to parallelise the execution of EDDL operations, including the parallelisation of the costly training operations; on the other side, it supports the most common hardware acceleration technologies, i.e., many-cores, GPUs and FPGAs, to further accelerate the training and inference operations in single computing nodes. Moreover, the DeepHealth HPC toolkit provides to the data/computer scientists the level of abstraction needed to describe the underlying computing infrastructure in a fully transparent way.

PREPRINT

[20] G. Guennebaud, B. Jacob and others, “Eigen v3.,” 2010. [Online]. A vailable: http://eigen.tuxfamily.org. [Accessed 2 November 2020].

[21] L. Dagum and R. Menon, “OpenMP: an industry standard API for shared-memory programming.,” Computational Science & Engineering, vol. 5, no. 1, pp. 46-55, 1998.