Reproducibility and Performance of Deep Learning Applications for Cancer Detection in Pathological Images

1st Christoph Jansen
CBMI – HTW Berlin and
Charité – Universitätsmedizin Berlin
Berlin, Germany
Christoph.Jansen@htw-berlin.de

2nd Bruno Schilling
CBMI – HTW Berlin
Berlin, Germany
Bruno.Schilling@student.htw-berlin.de

3rd Klaus Strohmenger
CBMI – HTW Berlin
Berlin, Germany
Klaus.Strohmenger@htw-berlin.de

4th Michael Witt
CBMI – HTW Berlin
Berlin, Germany
Michael.Witt@htw-berlin.de

5th Jonas Annuscheit
CBMI – HTW Berlin
Berlin, Germany
Jonas.Annuscheit@htw-berlin.de

6th Dagmar Krefting
CBMI – HTW Berlin
Berlin, Germany
Dagmar.Krefting@htw-berlin.de

***Abstract*—Convolutional Neural Networks (CNN) are used for automatic cancer detection in pathological images. These data-driven experiments are difficult to reproduce, because the CNNs may require CUDA-enabled Nvidia GPUs for acceleration and training is often performed on a large dataset stored on a researcher's computer, inaccessible to others.**

We introduce the RED file format for reproducible experiment description, where executable programs are packaged and referenced as Docker container images. Data inputs and outputs are described as network resources using standard transmission and authentication protocols instead of local file paths. Following the FAIR guiding principles, the RED format is based on and compatible with the established Common Workflow Language specification. RED files are interpreted by the accompanying Curious Containers (CC) software. Arbitrarily large datasets are mounted inside containers via FUSE network filesystems like SSHFS.

SSHFS is compared to NFS and a local SSD in artificial benchmarks and in the context of a CNN training scenario, where SSHFS introduces a performance decrease by a factor of 1.8.

We are convinced that RED can greatly improve the reproducibility of deep learning workloads and data-driven experiments. This is in particular important in clinical scenarios where the result of an analysis may contribute to a patient's treatment.

***Index Terms*—Reproducibility; Performance; Deep Learning; Machine Learning; Container; Docker; Filesystem in Userspace; CUDA; FAIR Principles; Common Workflow Language; Reproducible Experiment Descriptions; Curious Containers**

I. Introduction

Digital pathology addresses the storage and processing of pathological imaging data to enable use cases like automatic tumor detection in tissue samples. In this field, there is a rising interest for Deep Learning approaches, as they have the potential to outperform other computer vision algorithms [14].

But in medical decision support, methods reproducibility - the ability to obtain the same results when an image analysis task (also called virtual experiment) is reproduced under identical conditions - is a major issue. In this context, code, data and runtime environment have been identified as the critical components to achieve high reproducibility [18].

Containerized applications[1] are increasingly acknowledged to allow for reproducibility of a certain runtime environment [4], [7], [9]. The Common Workflow Language (CWL) allows for consistent description of commandline tools [2]. However, data and container image access descriptions, as well as runtime descriptions, are only partially possible using a CWL document. To be actually able to rerun an experiment, a sufficient description of all components must be provided, as well as tools to combine these information automatically and perform the replication.

In the data domain, the FAIR guiding principles are currently the de-facto standard for publishing scientific data [20]. The four principles findable, accessible, interoperable and reusable imply among others, that the data can be referenced by a global identifier (e.g. URL, DOI), is stored in open standard formats (e.g. CSV, JSON, PNG) and is accessible via common transfer and authentication protocols (e.g. SSH, HTTP(S) + Basic Auth). An appropriate license should clarify the terms of use and redistribution. These principles are applicable to code and, with the help of virtual machine or container filesystem images, can also be applied to runtime environments. With FAICE, we have introduced a tool-suite and ecosystem for reproducible virtual experiments [10]. In this context, each virtual experiment represents a single execution of a program with certain input files and arguments, but without any references to other experiments. This means, that workflows, where the outputs of one program are used as inputs to another program, are not supported. In contrast to typical workflow engines, the data transfer is handled

[1]https://www.docker.com/

by dedicated connector programs inside the container, such that input data is never transferred to a location outside the container and is automatically removed when the container is destroyed [11]. The system has been successfully employed for analysis of multidimensional biosignals [10]. The input data in that scenario are about 300 MB and are fetched directly through the REST-API of an online data management system [5].

In the realm of Deep Learning, specific technical challenges arise. The training of an artificial neural network (ANN) often requires very large datasets and make use of graphics processing units (GPU) to accelerate the training process. In digital pathology, so-called Whole Slide Images (WSI), have a very high resolution. WSI is the term for scanned and fully digitized slides, containing thin, stained tissue, prepared for microscopy. On the highest resolution level, a slide consists of approximately 200,000 x 100,000 pixels [3]. WSI have a typical file size of about 4 GB, but uncompressed slides - as required for image processing - inflate up to 50-60 GB each. A full training set typically exceeds 1 Terabyte of data. Therefore downloading a dataset via HTTP before running the algorithm is inefficient and might not be possible at all, if the executing computer does not provide the required storage space.

A common use case is the classifier training for cancer detection in pathological images. A prominent example in the digital pathology community is the CAMELYON16 challenge [8]. The task of the challenge was to predict if and where metastatic tissue is located in lymph nodes of women with breast cancer. The training set for the CAMELYON16 challenge consists of 270 WSI of lymph node tissue. As in previous challenges, convolutional neural networks (CNN) have proven to outperform other image analysis approaches. CNNs are a type of artificial neural networks (ANN) that are specifically suitable for image classification. The three main architecture ideas of CNNs are to use “local receptive fields, shared weights (or weight replication), and spatial or temporal subsampling“ [13]. A starting point for the popularity of CNN has been the winning of the ImageNet 2012 competition [12]. Since then, a lot of different CNN architectures have been developed, among them the freely available Inception-V3 model [17].

In this paper, we introduce Reproducible Experiment Descriptions (RED), a file-format that is built upon the commandline tool specification of CWL. It is a generic format for any virtual experiment based on a Linux commandline application and therefore not limited to Deep Learning applications or the presented biomedical use case. RED allows for descriptions of different container engines, including nvidia-docker to handle GPU-based workloads. Connector implementations to handle arbitrarily large data sets via network filesystems are introduced as well. Different file transfer mechanisms are compared regarding security and performance. Execution engines to interpret and run experiments were implemented as part of Curious Containers software project, including CC-FAICE as a successor of the FAICE tool-suite.

II. Methods

In this section we introduce the tools and components employed for the current study.

A. Data and Application

For the presented experiments, the CNN model Inception V3 is trained to classify pixels representing breast cancer in lymph node tissue. As training data set, the CAMELYON16 training set is employed.

1) Data: The CAMELYON16 training set has been introduced by the aforementioned challenge [8], and consists of 270 whole-slide-images (WSIs) and segmentation data as label for cancerous tissue. The original data is stored as compressed tif-files and an additional xml-file containing the segmentation. To make the data suitable as input for the CNN model training, the individual slides are split into tiles of 512x512 pixels each. To guarantee that the tiles do not only contain background or - in case of a slide that has been labeled as tumorous - not only normal tissue, only those tiles are stored that show at least 10% of the respective tissue. In normal slides, tissue areas are differentiated from background using the Otsu-thresholding algorithm, because segmentation data is not available in this case [16]. The segmentation results for tumorous and normal tissue are stored as a per-pixel bitmask alongside the RGB values of a tile. These tiles are stored in an individual HDF5 file for each slide. The HDF5 format is widely supported in the machine learning domain, because it provides an internal folder structure, can enable compression and is able to store numpy-arrays, which can be easily read by the CNN training code written in Python. As a further preprocessing step, a single-file training dataset is created from the individual HDF5 files. To avoid overfitting due to a training set with many tiles from few slides, tiles from each slide are randomly chosen and stored in one of four numpy-arrays: positive and negative samples for training and validation respectively. Therefore consecutive tiles in the resulting arrays are always from different slides. This sampling process is repeated 5000 times and results in a single HDF5 file, that reaches a file size of 1.3 TB.

2) CNN Training: The CNN training algorithm is implemented in Python3 using the Keras programming library for artificial neural networks [6]. Keras ships with a pretrained model of the *Inception V3* standard architecture available. Using a pretrained model significantly decreases the number of training epochs. Our benchmarks are therefore based on the Keras Inception V3 implementation, using Tensorflow as compute backend. To obtain consistent training performance benchmarks, each training is performed using the exact same settings. The training is limited to eight epochs, although the validation accuracy is not fully converged at this point. A common training strategy is to use so-called mini-batches. In each training step, another small subset of the training data is used as input. Keras supports mini-batches by employment of Python generators, such that batches are loaded when they are needed. Keras is able to execute data generators in the background while the CNN optimization is performed on a GPU. The generator uniformly selects a random starting position in one of the numpy arrays contained in the HDF5 file and loads the data as single 4-dimensional data chunk (tiles x width x height x RGB/bitmask). Every chunk is cropped down in the image dimensions from 512 x 512 pixels to 224 x 224 pixels at a random position. This is a data augmentation technique to further reduce overfitting, if the same tiles are used multiple times during the training process. Since array slicing can be implemented efficiently using numpy, only the cropped part of the data is read from disk or transferred via the network and only requires a single operation. The Python generator loads 100 tiles at once (ca. 19 MB). After loading the array slice, the bit mask is used to check if the cropped area of the tiles still fulfills the criterium of at least 10 % of the relevant tissue. This leads to a discard of about 25 % of the tiles. However, Keras manages the repetitive array slice loading in a way, that a new mini-batch is created when a new training step starts. The actual mini-batch size is set to 100 normal and 100 tumor tiles. Loading a mini-batch includes further data augmentation using rotation and mirroring operations. 50 mini-batch training steps are executed per epoch.

B. Common Workflow Language

As mentioned in the introduction, the Common Workflow Language (CWL) [2] provides a specification to describe input arguments, files and directories, as well as output files and directories of a commandline interface (CLI) application in a YAML file format. The description can be used by any execution engine that supports CWL to automate calls to a CLI application. However, the actual input data needs to be defined outside this description. For example, the official CWL reference implementation *cwltool*[2] can execute an application, if a CWL file and a JOB file are provided. The CWL file contains the generic interface description, that can be reused for any execution. The JOB file contains the specific file paths and arguments to be used for the current instantiation of the program. In order to reference files or directories in a JOB file, the *location* keyword can be used, that can either contain a path in the local filesystem or a URI to a network resource; for example starting with *http://* to indicate, that the HTTP protocol should be used. The handling of such network protocols is up to the CWL executing software and might not be portable between different CWL implementations. If any authorization is required, the credentials must be encoded in the URI string and proper handling of the credentials is again up to the given implementation. Unfortunately the JOB file cannot contain information on how to handle output files and directories. Besides the CLI specification, CWL provides additional specifications for workflows, that are not relevant to our use case.

The CWL CLI specification provides a *dockerRequirement* section, where a *dockerPull* keyword can be specified, to download a Docker image from a registry, which is then used to execute a CLI tool provided by the image in a Docker container. The CWL standard does not specify how to handle authorization credentials, which might be necessary to pull the image. If input files or directories are provided as JOB arguments, these files must be present on the host operating system and are provided inside the container by using Docker bind mounts. These are essentially filesystem mounts on top of the Docker image filesystem. This feature is implemented by *cwltool*. The *cwltool* developers decided to only start containers as a supposedly unprivileged user with the user and group IDs set to 1000. This requires an arbitrary user and group with this ID to be created as part of the container image build process and the CLI application included in the image must be executable by this user. The CC implementation follows this convention, such that container images created for CC are always compatible with *cwltool*.

C. Curious Containers

The Curious Containers (CC) software project[3] aims to to support reproducible experiments [10]. The main design criteria are, that (a) the applications are provided as Docker images, so-called appliances, (b) data is loaded directly into the appliance, (c) user authentication on the components like data storage and compute environment is supported, and that (d) all information required to describe the experiment is stored in a single file, that can be published and shared. Since it's first release, described in [10], the componentes have been completely redesigned to better fulfill the FAIR guiding principles and to be easier to employ. The new execution engines CC-FAICE and CC-Agency are described in subsection III-B.

D. Containerized Applications

Executable computer programs often require a complex set of dependencies, ranging from third party programming libraries, as well as interpreters and runtimes, to low-level APIs provided by the operating system. Ensuring compatibility of programs with other computers outside the original programming environment is therefore difficult. With container technologies like Docker, a program executable and all its dependencies, often including the userspace libraries of a certain Linux distribution, can be installed in a Docker image filesystem. Given the container image, the common interface for any containerized process is therefore reduced to a Linux kernel and a docker-engine to manage the container. There are two ways to publish such an appliance: On one hand, the Dockerfile, that serves as the building recipe for the container, can be published as source code, or the built Docker image can be uploaded to public Docker repositories. For all experiments, we provide both methods. A specific feature is the layered structure of Docker images. New images can be created based on existing images, making it easy to extend the appliance by specific features.

[2]https://github.com/common-workflow-language/cwltool

[3]https://www.curious-containers.cc/

1) Curious Containers support extension: The appliance must contain a set of tools and follow a certain structure to be compatible with CC. A system user with user and group ID 1000 must exist in the container image. The application must be installed as a commandline tool, that is included in the *PATH* environment and is executable by the aforementioned system user. These requirements conform to the CWL standard, such that appliances are compatible with any CWL runtime. In addition to the CWL requirements, the user must install any number of RED connector CLI programs. These connectors can then be referenced in a RED file to be invoked by the execution-engine to transfer data in and out of the appliance. In order to handle the execution of the various installed programs, CC mounts a management program called *agent* into the appliances. To successfully run the agent, a POSIX shell and a Python3 interpreter (version >= 3.4), which are part of most Linux distributions, must be present in the appliance.

2) CUDA support extension: Machine Learning algorithms are often based on programming frameworks like Tensorflow [1], which provides an abstraction layer for mathematical operations. Tensorflow can be used with CPUs, but can also take advantage of GPUs for accelerated performance. CUDA is a proprietary GPU compute interface and only compatible with Nvidia GPUs. Since Nvidia ships CUDA, as well as compatible hardware drivers, as closed source programs, Tensorflow can be downloaded in several flavors, each precompiled for a specific version of CUDA. Therefore a certain CUDA release is one of the dependencies of a Tensorflow machine learning application.

Nvidia-docker is a docker-engine plugin that makes Nvidia GPUs and its drivers available inside a running Docker container. It requires Docker and the official Nvidia hardware driver to be installed on the host. Using this plugin, a compatible CUDA version can be shipped as part of a Docker container image and the Tensorflow application can detect the GPUs. For improved resource management, a container can be configured to only have a subset of the GPUs accessible, if multiple graphic processors are available in the system.

E. Shared File Systems

As the main challenge is to load the data from the 1.3 TiB large file into the appliance, different data transfer strategies are investigated. To be able to employ the Python generators for subsequent data loading, shared file systems are considered, evaluated regarding availability in Linux containers and authentication methods.

1) Network File System: The NFS is a fundamental Linux/Unix technology to access a shared storage that is mounted into the local filesystem. The operating system kernel handles communication with the NFS server via remote procedure calls. Currently, version 2, 3, 4.0, 4.1 and 4.2 of the NFS protocol are available on modern Linux systems.

The NFS protocol versions support different security mechanisms to manage file and directory access on the shared filesystem. Access control can be disabled completely or be based on user and group IDs for all versions. Version 4.0 and onward support Kerberos based user authentication. Besides the encryption and signing of transferred data users are no longer identified by their ID in this mode, but with their respective Kerberos principal.

Like any other kernel filesystem, mounting an NFS share requires root privileges on the host system. It is therefore by itself not feasible for the presented use case, where data connections are performed by an unprivileged user inside a Docker container. Therefore we only use NFS in our benchmarks for performance comparisons.

2) Filesystem in Userspace: Filesystem in Userspace (FUSE) is a technology available for Linux and other operating systems, that enables filesystem drivers to run as a user process instead of kernel process. FUSE consists of a kernel module, as required by any filesystem, and *libfuse*, a userspace library. A filesystem implementation is linked against libfuse, to implement the various filesystem operations. As a consequence, FUSE filesystems can be mounted by a user without any administrator privileges on the system. If a user tries to access a directory mounted via FUSE, the syscall is handled by the FUSE kernel module, which in return communicates with the filesystem implementation via *libfuse*. This procedure requires multiple context switches between the kernel and the user space, which can cause performance degradations that are "completely imperceptible or as high as –83 %" [19], compared to in-kernel filesystems.

It is possible to mount FUSE filesystems as an unprivileged user inside a running Docker container. Unfortunately this is not possible in the fully isolated standard configuration; instead a container must be started with administrative privileges and access to the FUSE device using *–device /dev/fuse –cap-add SYS_ADMIN*. If the host uses AppArmor, *–security-opt apparmor:unconfined* disables this additional security layer for the given container. All of the mentioned options have potential security implications [15]. It means that running containers based on untrusted images should be avoided.

a) Considered FUSE filesystems: The flexibility of FUSE enables developers to implement a wide variety of filesytems, including the possibility to mount remote resources via network protocols. We have investigated three different network filesystems, namely SSHFS[4] (version 3.5.1), HTTPDirFS[5] (version 1.0.1) and FUSE-NFS[6] (git commit 91c69c5).

SSHFS is a mature filesystem implementation developed by the *libfuse* team and is included in most major Linux distributions. It provides file access via SSH and is therefore very flexible in terms of authentication. We use LDAP as a server-side authentication backend and central user management. File access authorization is provided by POSIX file-permissions corresponding to the user on the server side.

HTTPDirFS can mount HTTP(S) directory listings, as, for example, served by Apache2 and can use HTTP Basic Auth. It

[4]https://github.com/libfuse/sshfs
[5]https://github.com/fangfufu/httpdirfs
[6]https://github.com/sahlberg/fuse-nfs

is not a mature implementation and is not included with major Linux distributions. But as it fulfills the basic requirements, we have included it into our benchmarks.

Since we already use a network filesystem (NFS) with Kerberos authentication in our network, we investigated the usage of FUSE-NFS. Although FUSE-NFS supports NFS version 3 and 4, the only implemented authentication method is *sys*. This authentication method cannot be employed in the context of containers, because it is based on the IP address and the local user ID (NFS v3) or name (NFS v4). Therefore we excluded it from further analysis and benchmarks.

F. Hardware

For the experiments, a heterogeneous compute cluster consisting of 12 servers for CPU tasks, 3 servers for GPU tasks (GPU-S1, GPU-S2, GPU-S3) and 1 storage server is employed. The CPU servers are not used in our benchmark.

The storage server provides 188 GiB RAM and a RAID 5 disk array that consists of 16 10 TiB NLSAS HDDs with a dedicated hardware RAID controller. The disk array is formatted with an EXT4 filesystem and mounted into the operating system with a net capacity of 134 TiB. The installed CPUs are two Intel Xeon Gold 5122 with 4 cores / 8 threads at 3.60 GHz each. The GPU servers have 2.9 TiB of local NVME SSD storage (via PCIe) and 755 GiB RAM. Each GPU server provides 4 Nvidia Tesla V100 graphic cards with 16160 MiB of VRAM. The installed CPUs are two Intel Xeon Gold 6130 CPUs with 16 cores / 32 threads at 2.10 GHz. All servers are connected in a local network via two 10 Gb/s networking cards per server, that operate in link aggregation mode conforming to IEEE 802.1AX-2008.

G. Performance Benchmarks

We have performed two types of benchmarks. Firstly we measure file read speeds and compare the read speeds of a GPU server's (GPU-S1) local SSD to the network throughput when transferring file chunks from the storage server to a GPU server using different protocols. The second part of the tests involve the training of a CNN that is a IO heavy computation task under realistic conditions. While running the benchmarks, the servers were not used for any other tasks. Each training benchmark consists of exactly 9 test runs, which allows us to distribute the runs equally across 3 GPU nodes in parallel processing benchmarks.

III. Implementation

A. Reproducible Experiment Descriptions

Reproducible Experiment Descriptions (RED) are a YAML or JSON based file format to describe data-driven experiments in a way that they can be automatically replicated. In terms of the RED format, a data-driven experiment is executed by running a CLI application in a Docker container, that takes inputs (files, directories and CLI attributes) to produce outputs. The RED file structure is shown in YAML listing 1.

It contains the mandatory toplevel sections *cli* for the definition of the application itself, *inputs* for the definition

Listing 1: RED structure.

```yaml
redVersion: "6"
cli:
  # begin CWL
  cwlVersion: "v1.0"
  class: "CommandLineTool"
  baseCommand: ...
  inputs:
    inputA:
      type: integer
      inputBinding:
        position: 0
    inputB:
      type: Directory
      inputBinding: ...
  outputs:
    outputA:
      type: File
      outputBinding: ...
  # end CWL
inputs:
  inputA: 42
  inputB:
    class: Directory
    connector:
      command: "red-connector-sshfs"
      mount: true
      access:
        host: "data.example.com"
        username: "{{ssh_username}}"
        password: "{{ssh_password}}"
        dirName: "/data/set"
    listing: ...
outputs:
  outputA:
    class: File
    connector:
      command: "red-connector-ssh"
      access: ...
container:
  engine: "nvidia-docker"
  settings:
    image:
      url: "registry.example.com/x/y:z"
      auth: ...
    ram: 4096
    gpus:
      count: 1
execution:
  engine: "ccagency"
  settings:
    access:
      url: "agency.example.com"
      auth: ...
```

of the data locations and *container* for the description of the appliance. For remote execution of the experiment with cc-agency, the section *outputs* is also mandatory. The execution engine may be defined in the *execution* section. This is particularly useful, if the experiment should be executed in a certain infrastructure, for example due to specific requirements on the underlying system, as in our case the Nvidia plugin for the host's docker engine.

The *cli* section embeds the CWL file. Input is described in terms of a parameter values (inputA) or remote locations, as in inputB. The file transfer method is defined within the connector section, with further attributes dependent on the actual connector type. Please refer to the RED documentation for the full list of available connectors[7]. The listing shows the configuration for the sshfs-connector, that has been employed for the benchmarks. This is different from the ssh-connector used for the output files. The latter does not mount an external file system, but transfers a file via scp. The container section requires at least an engine and a URL of a container image. Currently docker and nvidia-docker are supported as engines. Further attributes like resource requirements can be added. Currently, two execution engines, CC-FAICE and CC-Agency, are supported (cf. sec. III-B). A RED file can contain engine specific settings under the *execution* keyword.

1) Credential Handling: As reproducible experiments require data to be accessible via standard network transmission protocols, authentication to restrict user access to sensitive data is crucial. Depending on the authentication method, a RED file would therefore contain confidential user credentials. To allow users to store, share or publish a RED file, any credentials can be replaced by variables. For example *password: "secret"* would be replaced by *password: "{{ssh_password}}"*. A RED execution engine detects these variables to fill in the missing information on demand, for example from a separate file containing the values provided by the user or via interactive commandline prompts.

Additionally, execution engines must avoid that any secret values are exposed, for example by writing them to a log file. We have therefore introduced protected keys that indicate that values must be handled with care. By default, only *password* is a protected key. If, for example, a given username should be protected as well, the user can prepend an underscore to the key (e.g. *username: "{{ssh_username}}"* becomes *_username: "{{ssh_username}}"*).

2) Batch Processing: As experiments often imply the execution of a certain CLI on many input files, RED supports batch processing. Instead of using the *inputs* and *outputs* keywords, the top level *batches* keyword can be used, that contains a list of inputs and outputs pairs (cf. lst. 2).

B. Curious Containers

Curious Containers has been adopted to implement the RED file format. To date, two different execution engines are available: CC-FAICE and CC-Agency. CC-FAICE is a collection of

[7]https://www.curious-containers.cc/

Listing 2: RED structure for batch processing.

```
redVersion: "6"
cli: ...
batches:
  - inputs: ...
    outputs: ...
  - inputs: ...
    outputs: ...
container: ...
execution:
  engine: "ccagency"
  settings:
    batchConcurrencyLimit: 9
```

commandline tools, including a basic RED execution-engine under the command *faice agent red*, that connects to a docker-engine on the local host. Other features of CC-FAICE are various tools to convert and validate RED files, as well as a tool called *faice exec* that reads the *execution* section of a RED file and invokes the specified engine. CC-Agency is a more comprehensive execution-engine to manage a cluster of Docker hosts. It exposes a REST API to receive a RED experiment and to provide information about the state and potential errors of scheduled experiments. CC-Agency can manage a heterogeneous cluster of CPU and GPU nodes, currently using a simple scheduling mechanism based on the RAM available on the compute nodes. Since the scheduling is asynchronous, experiments have to be written to a database, such that the database temporarily contains clear text credentials for the connectors. The values of these protected keys (cf. sec. III-A1) are deleted from the database, as soon as an experiment is finalized (entering one of the states *succeeded*, *failed* or *cancelled*). This implies that the current implementation of CC-Agency should only be used if the operator is trustworthy and the usage of temporary or one-time credentials is strongly recommended.

Fig. 1 summarizes the interactions between the experiment's components, all referenced in a RED file. The RED file is given to the execution engine, that communicates with a Docker engine to pull the appliance from a registry and to run the experiment in a container. Inside the running container, the *agent* management program (cf. sec. II-D1), handles the download or mounting of input data using connectors, invokes the app with CLI arguments and uploads output files to a storage server.

C. File Read Performance

To evaluate the file read performance, random 16 MiB data chunks are fetched from a single testfile. The read tests are run with the Flexible I/O tester[8]. For file access method we tested exclusive read access and simultaneous access by 9 processes.

[8]https://fio.readthedocs.io/

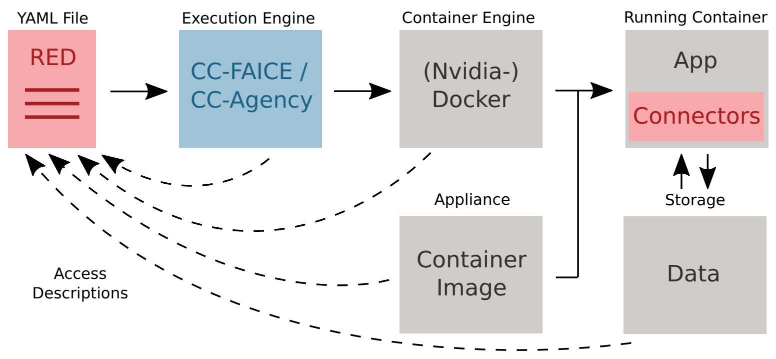


Fig. 1: Interaction between experiment components referenced in RED file.

All benchmarks are executed locally on GPU-S1, the testfile is stored locally (SSD) and on the storage server.

local SSD: The testfile is stored on the local NVMe SSD storage of GPU-S1 that is mounted through the operating system kernel.

NFS (sec=sys): The filesystem from the storage server is mounted via the NFS kernel driver to GPU-S1. System security is used for access control on the GPU-S1 side. It does not provide signing or encryption of transferred packages. NFS protocol version 4.0 is used.

NFS (sec=krb5p): This setup tests NFS filesystem with full Kerberos security mechanisms in place. The operating system kernel manages the NFS connection and manages authentication, signature and encryption of each data packet that is send or received. The NFS protocol version for this setup is 4.2.

SSHFS: The SSH-based filesystem driver utilizes FUSE do bind the remote share into the operating system. Similar to the krb5p setting of NFS, data transfer is fully signed and encrypted.

HTTPDirFS: This FUSE filesystem driver enables to mount a filesystem that is accessible via an HTTP-service (without SSL/TLS) into the operating system. We access the data through an Apache2 webserver running on the storage server with basic-authentication.

1) CNN Training Performance: For a detailed performance evaluation we included time measurements into the CNN training code. Anytime a generator starts or ends loading a training or validation mini-batch, a timestamp is saved. Using nvidia-docker, every training process was encapsulated in a Docker container with only one of four GPUs being visible to the containerized process.

The following benchmarks have been implemented:

SSD (sequential): The HDF5 dataset has been copied to the SSD of GPU-S1 beforehand. 9 separate docker processes are started consecutively using a shell script with the local directory mounted into the container as Docker volume.

NFS (sequential): The HDF5 dataset is located on the storage server and is remotely mounted via a Network File System (NFS) connection (using *sys* authentication without Kerberos) on the GPU-S2 host. This mounted directory is again mounted as a Docker volume into the container. Again, a shell script is used to run 9 consecutive tests.

SSHFS (sequential): For the SSHFS benchmarks CC-Agency is used as a cluster scheduler, where SSHFS is implemented as RED connector. The training is performed in separate containers for each run that are controlled by CC-Agency. SSHFS is not mounted on the GPU hosts beforehand, instead it is mounted directly in the running containers using FUSE. For the SSHFS sequential benchmark *batchConcurrencyLimit* is set to 1 (cf. sec. III-B), which means that CC-Agency only schedules a single test at a time.

SSHFS (3 parallel): These tests are executed with CC-Agency and *batchConcurrencyLimit: 3*, such that only 3 tests run in parallel at any given time. The tests are automatically spread across all GPU hosts.

SSHFS (9 parallel): These tests are executed with CC-Agency and *batchConcurrencyLimit: 9*. All tests run in parallel, with 3 tests per GPU node at a time. Each test is assigned to a different GPU.

Due to the weak performance (cf. sec. IV-A), HTTPDirFS has not been further considered.

The following evaluations have been performed for each benchmark:

Mean IO idle percentage: IO idle percentage for a single test run is calculated as the combined timespans where no batch loading happens divided by the total processing time multiplied with 100. The mean is calculated across all 9 runs of a benchmark.

Mean processing time in seconds: Mean total processing time of 9 test runs per benchmark.

Combined processing time in seconds: Processing time measured from the overall first timestamp to the overall last timestamp across all 9 runs of a benchmark. For some benchmarks the test runs are performed sequentially, while others are performed in parallel. In either case, all 9 test runs are always performed as a seamless set of tasks.

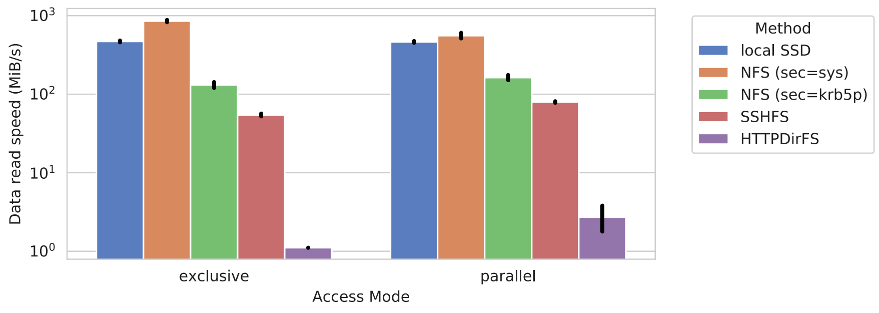


Fig. 2: Data read speed of random 16 MiB blocks compared by filesystem and exclusive/parallel access. NFS from a performant storage server is the fastest solution for both exclusive as well as parallel access, outperforming a local SSD. However, Kerberos authentication reduces the performance significantly. Note the logarithmic scale: HTTPDirFS is about 2-3 orders of magnitude slower than the other solutions.

IV. Results

A. File Read Performance

The results of the file read performance tests are shown in Fig. 2. Local data access is performed with a speed of ~470 MiB/s, with no significant performance loss during parallel access.

Compared to the local SSD, exclusive access to a file via NFS is 81 % faster with a mean speed of ~852 MiB/s. This results from the RAID 5 disk array and file caching mechanisms of the NFS kernel module on server and client side. However, it can be seen that parallel access of an NFS share significantly reduces the read speed down to ~581 MiB/s. This is still faster than the local SSD but 32 % slower than data read with exclusive access. Whereas NFS shows very promising performance it rigorously drops when available Kerberos security mechanisms are enabled. This can be explained by the overhead introduced through authentication, data encryption, signing, decryption and verification. This overhead requires additional CPU resources and network communication whereas no additional load on the RAID storage is introduced. This explains why no bandwidth drop between exclusive and parallel file access is observed. The filesystem on the storage server is fast enough to provide all data to all parallel requests, whereas the applied security mechanisms cause the limitation in bandwidth.

The measured performance of SSHFS is even lower than Kerberos-authenticated NFS. It can be assumed that performance declines because of the applied encryption throughout the whole communication as well as context switches introduced by the FUSE driver.

Most notably is the bad performance of HTTPDirFS. Even though the filesystem is mounted without encryption or any other security mechanisms except HTTP Basic Authentication, the mean data transfer rate is 1 to 1.1 MiB/s. Considering the overhead introduced by FUSE to be the same as with SSHFS and the lightweight protocol design of HTTP for downloading files (or chunks of files) we assume that the HTTPDirFS driver implementation is inefficient.

B. CNN Training Performance

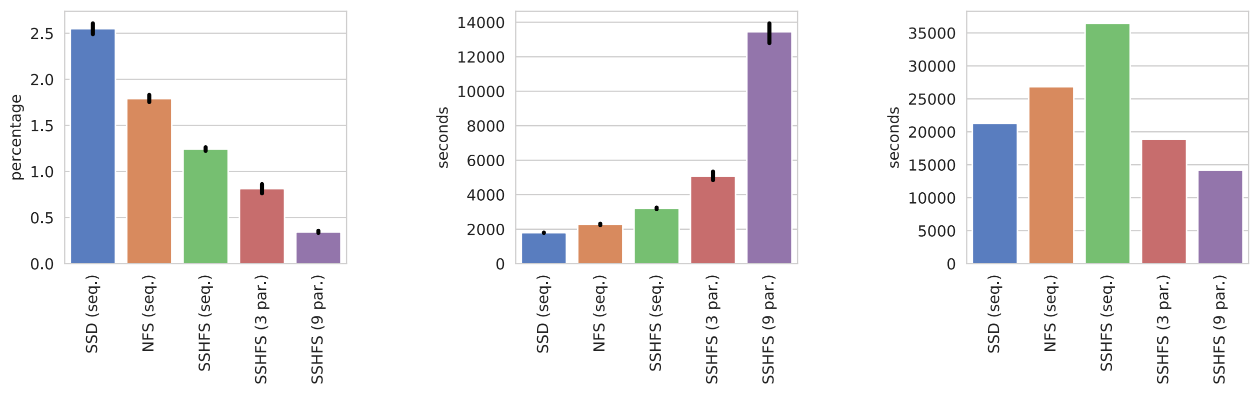
As described in section III-C1 three different measures have been obtained for each benchmark.

The IO idle ratio in Fig. 3a shows that basically all data transfers cause nearly full load. Even local access leaves only 2.5 % of the training time not used for loading array slices. As stated in section II-A2, Keras loads data in the background while the optimization process uses previously loaded mini-batches. It is difficult to discern, if the tested file access methods are already IO bottlenecks, but file access is definitely a critical factor for the performance.

Fig. 3b shows a notable increase of training time for SSHFS over SSD in sequential benchmarks. As expected, NFS performance better than SSHFS, but worse than a local SSD. This evaluation shows, that network based storage solutions clearly are a bottleneck for the given CNN training algorithm. In the SSHFS benchmark with 3 parallel trainings across all GPU nodes, the performance again decreases, because all processes compete for the resources of the same storage server. The factor compared to the SSHFS sequential test is 1.6. This factor is lower than 3 such that running parallel training still improves the combined training time by a factor of 1.9, as shown in Fig. 3c. The SSHFS benchmark with 9 parallel trainings shows no further performance gains, because the storage server is saturated with data requests. On the other hand, the storage server does not seem to be overloaded, because the combined training time of 9 parallel runs is not worse compared to 3 parallel runs.

V. Conclusion and Outlook

The RED format builds upon the CWL standard to describe data-driven experiments in a single-file format. We successfully implemented RED execution engines as part of the Curious Containers project to interpret and execute RED experiments. New features based on nvidia-docker and FUSE



(a) Mean IO idle percentage (n=9). (b) Mean processing time in seconds (n=9). (c) Combined processing time in seconds.

Fig. 3: Results of the CNN training performance benchmarks. Left: Mean IO idle time shows a very high load on all file access methods. Middle: The mean processing time of an individual training increases significantly by using SSHFS, in particular in parallel execution. Right: The overall processing time benefits from parallelization.

technologies were used to run fully containerized deep learning experiments, where arbitrarily large datasets are remotely mounted via network filesystems. However, a fully appliance-controlled data access using FUSE enabled SSH and HTTP comes with significant performance losses in data transfer rates. We believe, that referencing data sets via global URIs, with remote locations independent of the compute environment, greatly improves reproducibility, but trade-offs due to remote file access need to be considered in Deep Learning applications.

Training of the CNN Inception V3 model on the CAMELYON16 database has turned out to be a IO heavy task, where even SSD read speeds are saturated. Therefore we experienced a decrease in training performance, when the SSHFS filesystem driver implemented as a RED connector plugin was used. We expect but have not yet confirmed, that more complex CNN architectures, where computation time is a bottle neck, might suffer less from slowdowns due to file read speeds.

Future research encompasses investigation in more IO optimized network protocols like GridFTP, that is especially interesting because it provides separate management and data channels for authentication and file IO respectively, with exchangeable backend protocols for either channel. Other network protocols and potential caching techniques as well as Kerberos support for FUSE-NFS could improve performance and foster the use of decoupled data locations for improved reproducibility without compromises in authentication and authorisation.

VI. Published Experiments and Source Code

We have published the experiments, Dockerfiles and the application source code on Github[9]. We have created release versions for each Github repository and have archived them via Zenodo[10] to provide DOI links for reference. Docker images are hosted on Docker Hub[11]. The following resources are available online:

- **camelyon-utils**[12]: CLI tool to create HDF5 database of CAMELYON16.
- **camelyon-cnns**[13]: CLI tool to train CNNs on HDF5 database and measure batch load times.
- **appliances**[14]: Dockerfiles to create Docker images used in experiments.
- **camelyon-slide-to-tiles-experiments**[15]: Experiments to create HDF5 files per slide based on camelyon-utils.
- **camelyon-cnns-training-experiments**[16]: Experiments to run CNN trainings via SSHFS based on camelyon-cnns.

Nvidia-docker and FUSE are both supported in Curious Containers version 6.1 (available on Github[17]).

VII. Acknowledgments

This work is supported by the German Federal Ministry of Education and Research (project deep.TEACHING, grant number 01IS17056, project deep.HEALTH, grant number 13FH770IX6 and project BB-ITBoost, grantnumber 03FH0061X5).

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey

[9]https://github.com/deep-projects/

[10]https://zenodo.org/
[11]https://cloud.docker.com/u/deepprojects/
[12]https://doi.org/10.5281/zenodo.2581383
[13]https://doi.org/10.5281/zenodo.2581381
[14]https://doi.org/10.5281/zenodo.2581375
[15]https://doi.org/10.5281/zenodo.2581373
[16]https://doi.org/10.5281/zenodo.2581352
[17]https://github.com/curious-containers

Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Peter Amstutz, Michael R Crusoe, Nebojša Tijanić, Brad Chapman, John Chilton, Michael Heuer, Andrey Kartashov, Dan Leehr, Hervé Ménager, Maya Nedeljkovich, et al. Common Workflow Language, v1.0, July 2016. DOI: 10.6084/m9.figshare.3115156.v2.

[3] Peter Bandi, Oscar Geessink, Quirine Manson, Marcory van Dijk, Maschenka Balkenhol, Meyke Hermsen, Babak Ehteshami Bejnordi, Byungjae Lee, Kyunghyun Paeng, Aoxiao Zhong, et al. From detection of individual metastases to classification of lymph node status at the patient level: the camelyon17 challenge. *IEEE transactions on medical imaging*, 2018.

[4] Brett K. Beaulieu-Jones and Casey S. Greene. Reproducibility of computational workflows is automated using continuous analysis. *Nat. Biotechnol.*, 35(4):342–346, 04 2017.

[5] Maximilian Beier, Christoph Jansen, Geert Mayer, Thomas Penzel, Andrea Rodenbeck, René Siewert, Michael Witt, Jie Wu, and Dagmar Krefting. Multicenter data sharing for collaboration in sleep medicine. *Future Generation Computer Systems*, 67:466–480, February 2017.

[6] François Chollet. *Keras*. GitHub, 2015.

[7] Editors. Rebooting review. *Nat. Biotechnol.*, 33(4):319, Apr 2015.

[8] Babak Ehteshami Bejnordi, Mitko Veta, Paul Johannes van Diest, Bram van Ginneken, Nico Karssemeijer, Geert Litjens, Jeroen A. W. M. van der Laak, , and the CAMELYON16 Consortium. Diagnostic Assessment of Deep Learning Algorithms for Detection of Lymph Node Metastases in Women With Breast Cancer. *JAMA*, 318(22):2199–2210, 12 2017.

[9] Tristan Glatard, Rafael Da Silva, Nouha Boujelben, R. Adalat, Natacha Beck, Pierre Rioux, Marc-Etienne Rousseau, Ewa Deelman, and Alan Evans. Boutiques: an application-sharing system based on Linux containers. *Frontiers in Neuroscience*, 9, 2015.

[10] Christoph Jansen, Maximilian Beier, Michael Witt, Sonja Frey, and Dagmar Krefting. Towards reproducible research in a biomedical collaboration platform following the FAIR guiding principles. In *Companion Proceedings of the10th International Conference on Utility and Cloud Computing - UCC 17 Companion*. ACM Press, 2017.

[11] Christoph Jansen, Michael Witt, and Dagmar Krefting. Employing Docker Swarm on OpenStack for Biomedical Analysis. In Osvaldo Gervasi, Beniamino Murgante, Sanjay Misra, A.C. Ana Maria Rocha, M. Carmelo Torre, David Taniar, O. Bernady Apduhan, Elena Stankova, and Shangguang Wang, editors, *Computational Science and Its Applications : ICCSA 2016: 16th International Conference, Beijing, China, July 4-7, 2016, Proceedings, Part II*, pages 303–318. Springer International Publishing, Cham, 2016.

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[13] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[14] Geert Litjens, Thijs Kooi, Babak Ehteshami Bejnordi, Arnaud Arindra Adiyoso Setio, Francesco Ciompi, Mohsen Ghafoorian, Jeroen A.W.M. van der Laak, Bram van Ginneken, and Clara I. Sánchez. A survey on deep learning in medical image analysis. *Medical Image Analysis*, 42:60 – 88, 2017.

[15] Antony Martin, Simone Raponi, Théo Combe, and Roberto Di Pietro. Docker ecosystem – vulnerability analysis. *Computer Communications*, 122:30 – 43, 2018.

[16] Nobuyuki Otsu. A threshold selection method from gray-level histograms. *IEEE transactions on systems, man, and cybernetics*, 9(1):62–66, 1979.

[17] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[18] Rachael Tatman, Jake VanderPlas, and Sohier Dane. A practical taxonomy of reproducibility for machine learning research. Reproducibility in Machine Learning Workshop at ICML, 2018.

[19] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or not to FUSE: Performance of user-space file systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 59–72, Santa Clara, CA, 2017. USENIX Association.

[20] Mark D. Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E. Bourne, Jildau Bouwman, Anthony J. Brookes, Tim Clark, Mercè Crosas, Ingrid Dillo, Olivier Dumon, Scott Edmunds, Chris T. Evelo, Richard Finkers, Alejandra Gonzalez-Beltran, Alasdair J.G. Gray, Paul Groth, Carole Goble, Jeffrey S. Grethe, Jaap Heringa, Peter A.C 't Hoen, Rob Hooft, Tobias Kuhn, Ruben Kok, Joost Kok, Scott J. Lusher, Maryann E. Martone, Albert Mons, Abel L. Packer, Bengt Persson, Philippe Rocca-Serra, Marco Roos, Rene van Schaik, Susanna-Assunta Sansone, Erik Schultes, Thierry Sengstag, Ted Slater, George Strawn, Morris A. Swertz, Mark Thompson, Johan van der Lei, Erik van Mulligen, Jan Velterop, Andra Waagmeester, Peter Wittenburg, Katherine Wolstencroft, Jun Zhao, and Barend Mons. The FAIR Guiding Principles for scientific data management and stewardship. *Scientific Data*, 3:160018, March 2016.