

Optimising AI Training Deployments using Graph Compilers and Containers

Nina Mujkanovic
HPE HPC/AI EMEA Research Lab
Basel, Switzerland
nina.mujkanovic@hpe.com

Karthee Sivalingam
HPE HPC/AI EMEA Research Lab
Bristol, United Kingdom
karthee.sivalingam@hpe.com

Alfio Lazzaro
HPE HPC/AI EMEA Research Lab
Basel, Switzerland
alfio.lazzaro@hpe.com

Abstract—Artificial Intelligence (AI) applications based on Deep Neural Networks (DNN) or Deep Learning (DL) have become popular due to their success in solving problems like image analysis and speech recognition. Training a DNN is computationally intensive and High Performance Computing (HPC) has been a key driver in AI growth. Virtualisation and container technology have led to the convergence of cloud and HPC infrastructure. These infrastructures with diverse hardware increase the complexity of deploying and optimising AI training workloads. AI training deployments in HPC or cloud can be optimised with target-specific libraries, graph compilers, and by improving data movement or IO. Graph compilers aim to optimise the execution of a DNN graph by generating an optimised code for a target hardware/backend.

As part of SODALITE (a Horizon 2020 project), MODAK tool is developed to optimise application deployment in software defined infrastructures. Using input from the data scientist and performance modelling, MODAK maps optimal application parameters to a target infrastructure and builds an optimised container. In this paper, we introduce MODAK and review container technologies and graph compilers for AI. We illustrate optimisation of AI training deployments using graph compilers and Singularity containers. Evaluation using MNIST-CNN and ResNet50 training workloads shows that custom built optimised containers outperform the official images from DockerHub. We also found that the performance of graph compilers depends on the target hardware and the complexity of the neural network.

Index Terms—MODAK, SODALITE, HPC, cloud, performance optimisation, AI training, Singularity container, graph compilers

I. INTRODUCTION

Increasing availability of data and the computational power of High Performance Computing (HPC) have driven the adoption of Artificial Intelligence (AI) in recent years. Deep Learning (DL), a subset of AI that uses multi-layers of neural networks to progressively extract higher level features from raw data, has dramatically improved the state-of-the-art in domains like speech recognition, visual object recognition, and many others [1], [2]. This growth started with the success of the Convolutional Neural Network (CNN) [3] *AlexNet* [4] in 2012. *AlexNet* is computationally expensive and used GPUs to accelerate the training time. Frameworks like *TensorFlow* [5], *PyTorch* [6], *MXNet* [7], and *CNTK* [8] simplify the development and deployment of DL training workloads. Some frameworks also support a high-level language like *Keras* [9].

In recent years, DL training networks have grown in size and complexity. With exponential increase in data and

use cases, AI training workloads are being deployed across heterogeneous hardware targets like HPC and cloud. The user experiences in cloud environments and on HPC systems differ vastly. Cloud environments offer a number of Domain Specific Language (DSL) based tools such as *Terraform* [10] and *Cloudify* [11] to simplify the management of the entire application life-cycle, including the deployment, monitoring, and maintenance of application models. HPC systems on the other hand require specialist knowledge of the system and command line tools to manage the application lifecycle. They are accessed using tools such as Secure Shell (SSH), and require job submission to compute nodes via workload managers like SLURM [12] and TORQUE [13]. This can be a high hurdle for domain scientists interested in running experiments in HPC systems compared to cloud.

The cross-section of developing, optimising, and deploying AI applications across heterogeneous infrastructures like HPC or cloud environments poses a complex problem. EU projects of the Heterogeneity Alliance [14] like COLA [15], TANGO [16], HiDALGO [17], Exa2Pro [18], EcoScale [19] and SODALITE aim to deliver software tools, methods, and knowledge to enable next-generation applications to use heterogeneous hardware. SODALITE [20], a European Horizon 2020 project, aims to solve the problem of deploying workloads across heterogeneous environments by providing tools for software-defined infrastructures. In SODALITE, we have developed MODAK, a model-based application deployment optimiser for static optimisation in software defined infrastructure. In this paper we introduce MODAK and evaluate its usage for optimising AI training workloads using graph compilers and containers.

II. RELATED WORK

Application performance and scalability are important for HPC users. The optimisation process generally involves manual profiling and tuning of application parameters to suit target hardware. Furthermore, it is not portable and needs to be repeated when moving to other HPC systems due to the diversity of hardware in HPC systems.

The automation of application optimisation on both HPC and cloud systems requires models that can be used for performance prediction and to study how different hardware components affect performance, a task made more complex

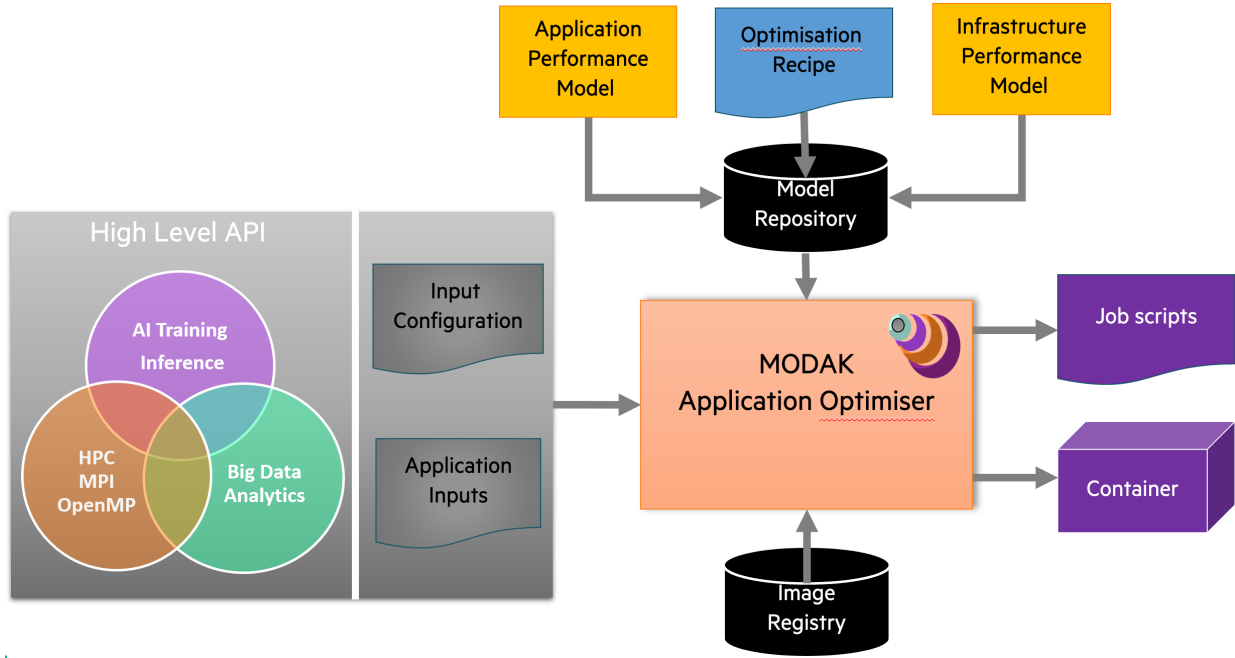


Fig. 1. MODAK architecture

by the wide variety of cloud offerings available with a wide variety of hardware. Application profiling and historical data gathered on HPC and cloud systems were used by [21] to create a performance model. *ParaOpt* [22], a tool that auto-tunes application configurations for different instance types based on runtime and cost, was evaluated for genomics, molecular dynamics, and machine learning applications on multiple public clouds.

A number of works explored the performance of cloud environments. Exabyte compared cloud targets using the Linpack benchmark [23], and developed a software tool for the continuous evaluation of various cloud environments [24]. EPCC directly compared the performance of HPC on-premise systems and the Oracle cloud cluster using the DiRAC application benchmarks [25], discovering issues in the usability and scalability of cloud based clusters.

Many tools are developed to optimise application deployments that are packaged as containers. *ConfAdvisor* [26] is a tuning framework for containers on Kubernetes. AWS compute optimiser [27] optimises workloads for both cost and performance based on historical utilization metrics. Google [28] similarly offers optimised containers for AI application deployments on the Google Cloud Platform. *HPAI* project [29] studied the feasibility of deploying AI workloads in HPC systems using *Charliecloud* [30].

In the next section, we will introduce MODAK, an application optimiser for software defined infrastructures like HPC and cloud.

III. MODAK

In a software defined world, where software rules and hardware is abstracted, enabling applications to optimally run

on diverse targets gives flexibility and saves money and time. SODALITE [20], a European Horizon 2020 project, aims to solve the problem of deploying workloads across heterogeneous environments by providing tools for software-defined infrastructures (SDI) that place the computing infrastructure under software control, abstracting away hardware dependencies. This simplifies and improves user exploitation of heterogeneous targets like HPC clusters, cloud environments, and edge devices.

MODAK is the SODALITE component responsible for enabling static optimisation of application deployment in a software defined way. The performance of an application when deployed in a specific infrastructure can be predicted using performance models of the application and infrastructure. The performance models are developed by running standard benchmarks across different configurations of both the application workload and the deployment infrastructure, and then building a linear statistical model. This model informs MODAK about how the application parameters, such as the input data size and format, affect the performance relative to the performance characteristics of the target infrastructure, such as peak performance and memory bandwidth. Using this knowledge, MODAK maps the optimal application parameters to the infrastructure target and builds an optimised container.

Figure 1 shows the MODAK architecture and its dependencies. Application performance optimisation is highly dependent on the application, its configuration, and the infrastructure. MODAK supports three major application types for static optimisation - AI training and inference, Big Data Analytics, and traditional HPC. The data scientist, or AoE, selects application optimisations using the SODALITE IDE.

Optimisations include changes to the application configuration, the environment, or the runtime. Application runtime parameters can be further autotuned for improved application performance.

In order to apply the optimisations, the Application Optimiser requires that application code be written in a standard high-level API, along with the application inputs and configuration. This enables the Optimiser to make performance decisions based on the available target. The Optimiser uses the pre-built, optimised containers from the Image Registry and modifies them to build an optimised container for the application deployment. The Application optimiser also makes changes to runtime, deployment, and job scripts for submission to HPC schedulers. Describing all the components of MODAK is out of scope of this paper and a detailed description can be found in [31].

IV. BACKGROUND

This subsection provides a brief contextual overview of the container solutions, AI frameworks, and graph compilers that are modelled for AI training optimisation for MODAK.

A. Container technology

Containers are a technology with roots in the Unix chroot command released in 1979 [32]. Linux containers (LXC), a precursor to the below listed technologies, use OS-level virtualization to isolate processes and resources in separate user namespaces [33], a virtualization technique with a far lower overhead and higher scalability than hypervisor virtualization [34].

Docker [35] performs virtualization using LXC for kernel-level namespace isolation and cgroups for resource control. While it is an industry standard in cloud environments, its design poses security and performance issues on HPC systems. In particular, it does not support multi-user HPC systems, and its use of root daemons to build and run containers enables users to gain privileged access to the host systems network filesystem.

Shifter [36] and *Charliecloud* [30], developed at NERSC and LANL respectively, were both designed with HPC systems in mind, making them more suitable to traditional HPC workflows. *Charliecloud*, in particular, is a lightweight containerization technology based on a User Defined Software Stack (UDSS) and developed around the strict security requirements posed by sites. Drawbacks include a high administrative overhead (*Shifter*) and a current lack of community uptake (*Charliecloud*).

Singularity [37] [38], developed by Berkeley National Laboratories, appears a good compromise between the cloud standard *Docker* and the HPC-specific *Shifter* and *Charliecloud*. Built for HPC systems, and offering native support for HPC components including resource managers (*Slurm*, *Torque*, etc.), job schedulers, and some MPI features, it also offers an easy containerization workflow for users. Its privilege model relies on SUID and non-privileged user namespaces to launch containers as child processes, thus allowing for non-root users

to create and launch containers safely. *Singularity* can import and run *Docker* images directly.

We chose *Singularity* as the optimal solution container deployment, with plans to extend to *Docker*. Further reading on container technologies can be found in [39], [40].

B. Graph compilers

An approach all AI frameworks have in common is the use of intermediate representations (IR) to represent the neural network models as computational graphs, with nodes representing tensor operations and edges the data dependencies between them. There are typically multiple levels of IR, with high-level IRs residing in the frameworks' user-facing front-end, and low-level IRs residing in the back-end.

A set of framework specific compilers can be used to perform optimisations on the generated graph IRs. These graph compilers can be grouped into two types - low-level tensor compilers, focused on the construction of high-performance operators for compute intensive operations, and deep learning compilers, focused on high-level optimisations on the IR followed by offloading to vendor specific libraries.

XLA (Accelerated Linear Algebra) [41] [42] is a TensorFlow specific graph compiler that accelerates linear algebra. It accepts a graph defined in the High Level Optimiser (HLO) IR and performs target-independent optimisation and analysis on it, such as operation fusion and buffer analysis. The optimised HLO IR is then sent to the back-end, which performs further HLO-level optimisations targeted to the hardware. The final code is generated using LLVM.

GLOW [43], short for graph lowering, optimises PyTorch models by lowering the graph into a two-phase IR, with strong focus on the low-level IR. The high-level IR is then used for domain-specific optimisations, while the low-level instruction-based, address-only IR is used to perform memory-related optimisations. Machine specific code is generated at the lowest level.

nGraph [44] is a framework independent graph compiler that can be used by TensorFlow, PyTorch, and a number of other frameworks. It acts as a bridge, porting the framework specific graph model to a common, intermediate high-level IR that is then used to generate code optimised for a specific back-end via vendor-specific libraries.

V. METHODOLOGY

In this section, we demonstrate the usage of MODAK for optimising AI training workloads using containers and graph compilers.

A. MODAK AI Training Example

Figure 2 shows an example usage of MODAK for AI training application deployment. A data scientist uses the SODALITE IDE to build the application model with input data, configuration, and optimisations. The application, written in a high-level language or API like Keras or *ONNX* [45], is then deployed in a *Docker* or *Singularity* container with the selected AI framework, using optimised libraries like *MKL*

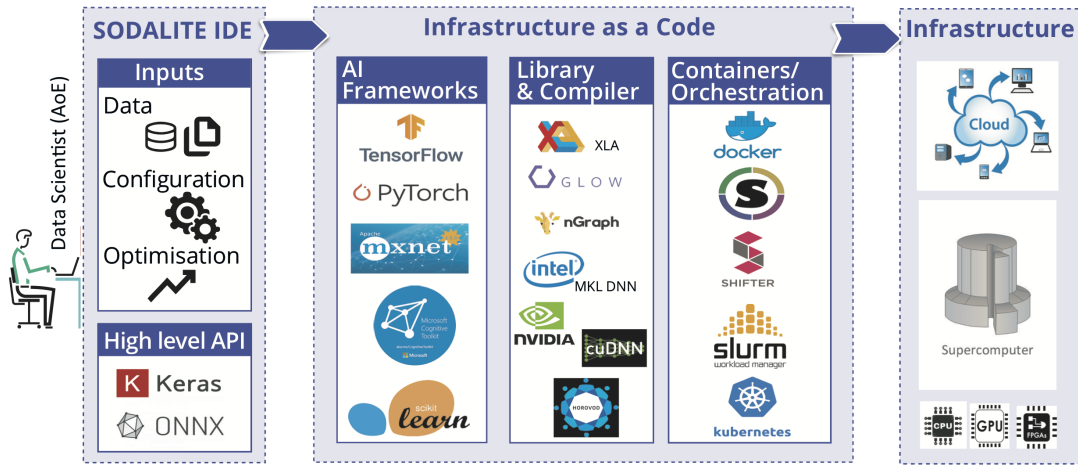


Fig. 2. SODALITE AI example

[46] or *cuDNN* [47], and compilers like *XLA* and *GLOW*. This optimised, containerised application is then deployed to an HPC or cloud system.

In the IDE, the data scientist encodes optimisation options in a *json* format, which is input to MODAK. Listing 1 shows a section of an optimisation DSL for a TensorFlow deployment. Here, custom build optimisations for a selected target (*x86* and *NVIDIA*) and the *XLA* compiler are enabled. MODAK prebuilds TensorFlow containers and tags them based on supported optimisations. Based on the selected optimisations in the DSL, MODAK selects the optimised container. MODAK can also build a container during deployment.

Listing 1. Example Optimisation DSL

```
"optimisation": {
  "enable_opt_build": true,
  "app_type": "ai_training",
  "opt_build": {
    "cpu_type": "x86",
    "acc_type": "Nvidia"},
  "ai_training": {
    "tensorflow": {
      "version": "1.1",
      "xla": true }}}}
```

B. Containers for AI frameworks

For MODAK, we created and evaluated AI framework containers on the SODALITE HPC testbed set up at *HLRS*, the research and supercomputing center affiliated to the University of Stuttgart [48]. The testbed consists of a front-end node running Torque, and five compute nodes, each hosting an Nvidia GeForce GTX 1080 Ti GPU, an Intel(R) Xeon(R) CPU E5-2630 v4 processor, and 125GB of main memory. All nodes allow access via *https*.

The container runtime installed on the system is Singularity version 3.4.1-1. In order to build and run containers, an administrator has to add additional mappings to the *Singularity* user

TABLE I
SOURCE OF AI FRAMEWORK CONTAINERS.

AI Framework	version	Hub	pip	opt-build
TensorFlow	1.14		X	X
TensorFlow	2.1	X	X	X
PyTorch	1.4	X	X	X
MXNet	1.6	X		
CNTK	2.7	X		
XLA	2.1	X	X	X
GLOW	NA			X
nGraph	1.14		X	

namespace UID and GID files to enable the use of *fakeroor*, a feature used to impersonate root without superuser escalation [49].

We determined which frameworks and graph compilers to benchmark based on popularity, availability of images, clarity of build instructions and documentation. Table I lists the set of AI framework images, graph compilers, versions, and their source. Official project images were downloaded from DockerHub (Hub) or packaged as Singularity containers using the Python package manager (*pip*) or by following the source build instructions available on the project websites (opt-build). We ensured that optimisation libraries were available and matched across the official and our images (e.g. TensorFlow *XLA*, etc.). Note that, while *XLA* is listed separately in the table, it is auto-built as part of the TensorFlow framework. For *XLA* and *nGraph*, the supported TensorFlow version is specified.

The DockerHub containers were retrieved using the *Singularity* pull command, which directly ports *Docker* containers to Singularity *.sif* files. We chose the images tagged with the required *version* and a *cpu* or *gpu* target to establish a baseline for comparison.

To build the remaining containers, we created Singularity definition files. The definition files are composed of a header that describes the operating system (OS) used within

the container, and multiple sections for pre-build setup, file importation into container, container environment setup, post OS installation container commands, etc.

As dependencies differ depending on whether containers execute CPU or GPU workloads, we developed two base OS containers to be called on in the definition header and used by our custom built containers.

C. Containers for CPU

The CPU enabled containers use a custom built Ubuntu 18.04 image as the base OS in the header. The image includes the packages *llvm-8*, *clang-8*, and *Python3*. The remaining build instructions are encoded in the post OS installation section. For the pip based containers, this includes adding commands that install additional dependencies, followed by the *pip* command to install the framework or compiler as per project instructions.

Similarly, the source build containers have all installation instructions encoded in the post section and adhere to the instructions given by the individual projects' documentation [50]–[52]. Where applicable, compiler optimisation flags were set to improve performance on the CPU. TensorFlow, specifically, uses the build tool *Bazel*, which accepts compiler flags via the argument `--copt`.

D. Containers for GPU

The GPU containers use NVIDIA DockerHub images containing the *NVIDIA-kernel*, *cuda toolkit 10.1*, *cudNN7*, and Ubuntu 18.04 as the base OS. We chose the NVIDIA base image to avoid portability issues and ease dissemination, as it is not possible to retrieve *cudNN7* via the command line. All NVIDIA package paths are then set in the container environment section to enable their use for source builds.

The remainder of the container build file is similar to that of the CPU containers. The source build commands are set in the post section of the file, with changes made to reflect differences in GPU builds, especially pertaining to the TensorFlow build.

All containers - CPU and GPU - are then built using the *Singularity* build command with the `--fakeroot` flag set. Depending on the framework or graph compiler, this build can take from a couple of minutes to multiple hours. The built containers can then be run interactively, or launched to execute specific commands. Note that *Singularity* places strong restrictions on GPU containers - the container must have the *nvidia-kernel* and any other dependencies such as *CUDA* installed, and the *nvidia-kernel* version must match the host *nvidia-kernel* version. This requirement can be circumvented by using the *Singularity* NVIDIA flag `--nv` when launching containers.

E. Benchmarks

We measured container performance by performing image classification training and timing the execution of a set number of epochs. To properly assess the frameworks on both CPU and GPU, we chose to train on two datasets, *MNIST* [53] for the CPU workload, and *ImageNet* [54] for the GPU workload.

MNIST training is an image classification problem for handwritten digits [55] [56]. *MNIST* itself refers to a dataset of 60,000 gray-scale images containing handwritten digits from 0 to 9.

ImageNet is a database that consists of more than 14 million hand-annotated images in 20,000 categories. The dataset, published in 2009, eventually evolved into the annual ImageNet Large Scale Visual Recognition Competition (*ILSVRC*) [57], at which *AlexNet* [4] achieved a novel 15.3% top-5 error rate in 2012, setting off a longterm trend of running DL workloads on GPUs.

The *MNIST* dataset was trained on a CNN consisting of a combination of two convolutional layers, two *maxpool* layers, two fully connected layers, and a *softmax* activation function. For all benchmarks, we used a batch size of 128, image size of (28,28), and trained the network with 1,199,882 trainable parameters for 12 epochs.

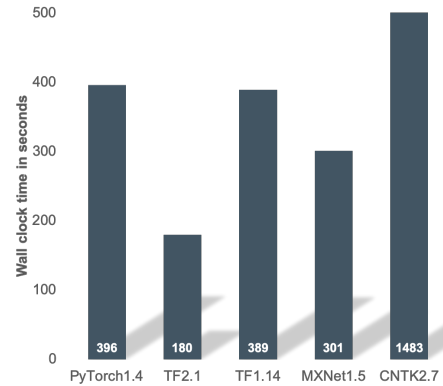


Fig. 3. Performance of various AI framework containers on CPU MNIST training workload

For the *ImageNet* training, we used the *ResNet50* [58] residual network, a fifty-layer deep neural network. Residual networks use skip connections to pass residual functions, minimizing the problem of vanishing or exploding gradients [59] and permitting the use of far deeper networks. For all *ResNet50* benchmarks, we used single precision, a batch size of 96, and trained for 3 epochs.

We wrote the *MNIST* and *ImageNet* workloads using the DSL of each framework and graph compiler. The workloads were submitted to one node exclusively per job using a *Torque* submission file. The training was not performed to convergence but instead stopped after 12 epochs for *MNIST*, and 3 epoch for *ImageNet*. This considerably shortened training time as we had determined in previous experiments that the main overhead occurred during the first epoch, while timing results for all remaining epochs remained stable.

VI. RESULTS

In the section, we discuss initial results generated by our benchmarks. In all figures, the Y-axis denotes the wallclock time in seconds required to complete 12 epochs of *MNIST* training, or the average time per epoch of *ResNet50* training.

Also *TFx.x* refers to TensorFlow version *x.x*, *src* post-fix refers to the optimal source built container and *XLA* or *NGRAPH* post-fix denotes is that graph compiler is enabled. PyTorch version 1.4 is used for all comparisons.

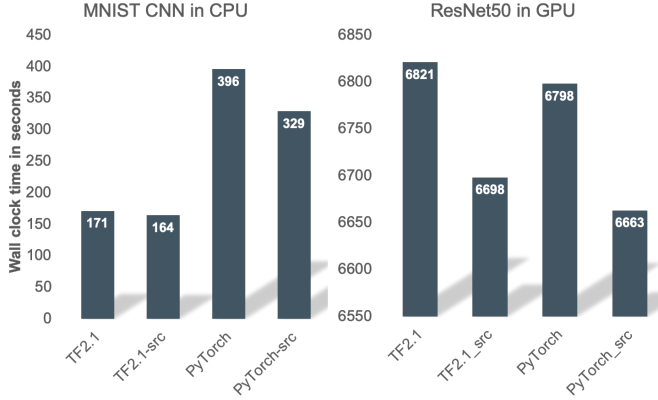


Fig. 4. Performance of AI framework containers custom built from source. Left - MNIST CNN training in CPU only. Right - ResNet50 in GPU.

The results of Figure 3 show the performance of the DockerHub containers for *MNIST* CNN training on CPU only. Graph compilers are not enabled for these results. As can be seen, TensorFlow 1.14, PyTorch, and MXNet (v1.6) perform similarly across the board, with the exception of TensorFlow 2.1, which shows a nearly 54% improvement over TensorFlow 1.14, likely due to eager execution being enabled by default starting from TensorFlow 2.0, while TensorFlow 1.14 uses graph execution. CNTK (v2.7) is a far outlier due to a lack of CPU optimisations, as mentioned in the official documentation. Note that MXNet and CNTK were only evaluated for comparison purposes and no further containers were evaluated beyond those attained from DockerHub.

Figure 4 (left) compares the training time of MNIST CNN in custom built containers to that of DockerHub containers. The TensorFlow custom built container shows little improvement (4%) compared to the DockerHub container, whereas the PyTorch container gives a substantial 17% speedup over the official DockerHub one.

Figure 4 (right) shows the result of ResNet50 training on *ImageNet* data with a custom built AI framework for the NVIDIA GPUs. A slight 2% performance improvement is visible for both TensorFlow and PyTorch source built containers. We see a similar performance for MXNet containers.

Figure 5 (left) compares the performance of TensorFlow in combination with *XLA* and *nGraph* for MNIST CNN training on a CPU. *XLA* is supported in most versions of TensorFlow, whereas *nGraph* is not yet supported for TensorFlow 2.x. We evaluated *XLA* on the latest standard release version 2.1, and *nGraph* on TensorFlow 1.4. A marked performance loss can be observed when running TensorFlow with *XLA* on the CPU. This is likely due to the fact that the *XLA* team has in recent years focused exclusively on the optimisation of just in time compilation on GPUs, and the overhead induced by the additional graph compilations on a simple network like

MNIST [42]. *nGraph*, on the other hand, shows speedup with a 30% improvement.

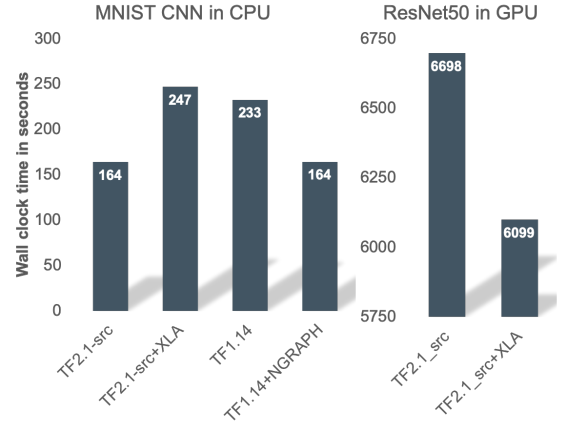


Fig. 5. Performance of AI framework containers with graph compilers. Left - MNIST CNN training in CPU only. Right - ResNet50 in GPU.

Figure 5 (right) shows the performance of the TensorFlow 2.1 source build using *XLA* on the ResNet50 for ImageNet workload on a GPU. The performance is improved by 9% using *XLA*, which is significantly better than the 30% performance degradation seen on the CPU.

We are currently evaluating the *GLOW* graph compiler container on CPU and GPU, and the *nGraph* graph compiler on the GPU. Thus far, the benchmark results show a trend of custom-built, hardware-targeted containers improving performance. For graph compilers, the optimisation results are dependent on the neural network and the target hardware.

VII. CONCLUSIONS

Software-defined infrastructures offer flexibility in deploying AI training workloads across heterogeneous targets. With diverse hardware, optimising AI workloads with different configurations and data sets is mandatory. We introduced MODAK, a novel tool that maps optimal application parameters to infrastructure using performance modelling and container technology. We demonstrated the usage of MODAK by optimising AI training deployments with graph compilers and Singularity containers. We showed an up to 17% speedup using custom built optimised containers and up to a 30% speedup using graph compilers. We also found use cases where graph compilers slowed the training. These benchmark results are then used to model the performance and optimise similar training workloads.

FUTURE WORK

Additional work is required to benchmark *GLOW* and *nGraph* on PyTorch and MXNet, as well as any emerging compilers. We plan to extend our containers to encompass more HPC specific workloads, such as MPI applications. Finally, while Singularity containers are the preferred runtime for HPC, we will port the images to Docker for easier dissemination of the project.

ACKNOWLEDGMENT

This work has received funding from the European Unions Horizon 2020 research and innovation program under grant agreement No 825480. We would like to acknowledge colleagues at the HPE HPC/AI EMEA Research lab and SODALITE for their support. We also thank *Tim Dykes* for proof-reading this paper and *Irene Ferrario* and *Harvey Richardson* for the support and guidance.

REFERENCES

- [1] M. Z. Alom, T. M. Taha, C. Yakopcic, S. Westberg, P. Sidike, M. S. Nasrin, B. C. Van Esesn, A. A. S. Awwal, and V. K. Asari, "The history began from alexnet: A comprehensive survey on deep learning approaches," *arXiv preprint arXiv:1803.01164*, 2018.
- [2] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.
- [3] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological cybernetics*, vol. 36, no. 4, pp. 193–202, 1980.
- [4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.
- [5] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.
- [6] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems* 32, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: <http://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>
- [7] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.
- [8] F. Seide and A. Agarwal, "Cntk: Microsoft's open-source deep-learning toolkit," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 2135–2135.
- [9] F. Chollet *et al.*, "Keras," <https://keras.io>, 2015.
- [10] 2020. [Online]. Available: <https://www.terraform.io>
- [11] 2020. [Online]. Available: <https://cloudify.co/>
- [12] M. A. Jette, A. B. Yoo, and M. Grondona, "Slurm: Simple linux utility for resource management," in *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*. Springer-Verlag, 2002, pp. 44–60.
- [13] D. Klusáček, V. Chlumský, and H. Rudová, "Planning and optimization in torque resource manager," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, 2015, pp. 203–206.
- [14] 2020. [Online]. Available: <http://heterogeneityalliance.eu>
- [15] 2020. [Online]. Available: <https://project-cola.eu>
- [16] 2020. [Online]. Available: <http://www.tango-project.eu>
- [17] 2020. [Online]. Available: <https://hidalgo-project.eu>
- [18] 2020. [Online]. Available: <https://exa2pro.eu>
- [19] 2020. [Online]. Available: www.ecoscale.eu
- [20] Sodalite. <https://www.sodalite.eu/>, last accessed 10 June 2020.
- [21] M. Baughman, R. Chard, L. T. Ward, J. Pitt, K. Chard, and I. T. Foster, "Profiling and predicting application performance on the cloud," in *UCC*, 2018, pp. 21–30.
- [22] C. Wu, T. Summer, Z. Li, A. Woodard, R. Chard, M. Baughman, Y. Babuji, K. Chard, J. Pitt, and I. Foster, "Paraopt: Automated application parameterization and optimization for the cloud," in *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2019, pp. 255–262.
- [23] M. Mohammadi and T. Bazhiron, "Comparative benchmarking of cloud computing vendors with high performance linpack," in *Proceedings of the 2nd International Conference on High Performance Compilation, Computing and Communications*, 2018, pp. 1–5.
- [24] —, "Continuous evaluation of the performance of cloud infrastructure for scientific applications," *arXiv preprint arXiv:1812.05257*, 2018.
- [25] 2020. [Online]. Available: <https://www.epcc.ed.ac.uk/blog/2020/06/benchmarking-oracle-bare-metal-cloud-dirac-hpc-workloads>
- [26] T. Chiba, R. Nakazawa, H. Horii, S. Suneja, and S. Seelam, "Confadviser: A performance-centric configuration tuning framework for containers on kubernetes," in *2019 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2019, pp. 168–178.
- [27] Aws compute optimizer. <https://aws.amazon.com/compute-optimizer/>, last accessed 12. June 2020.
- [28] S. Krishnan and J. L. U. Gonzalez, "Google compute engine," in *Building your next big thing with Google cloud platform*. Springer, 2015, pp. 53–81.
- [29] D. Brayford, S. Vallecorsa, A. Atanasov, F. Baruffa, and W. Riviera, "Deploying ai frameworks on secure hpc systems with containers," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019, pp. 1–6.
- [30] R. Priedhorsky and T. Randles, "Charliecloud: Unprivileged containers for user-defined software stacks in hpc," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–10.
- [31] K. Sivalingam *et al.*, "Prototype of application and infrastructure performance models," in *SODALITE Deliverables*. EC, 2020. [Online]. Available: <https://www.sodalite.eu/deliverables>
- [32] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.
- [33] M. Helsley, "Lxc: Linux container tools," *IBM developerWorks Technical Library*, vol. 11, 2009.
- [34] A. M. Joy, "Performance comparison between linux containers and virtual machines," in *2015 International Conference on Advances in Computer Engineering and Applications*. IEEE, 2015, pp. 342–346.
- [35] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.
- [36] L. Gerhardt, W. Bhimji, M. Fasel, J. Porter, M. Mustafa, D. Jacobsen, V. Tsulaia, and S. Canon, "Shifter: Containers for hpc," in *J. Phys. Conf. Ser.*, vol. 898, 2017, p. 082021.
- [37] Singularity. <https://sylabs.io/singularity/>, last accessed 10. June 2020.
- [38] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PloS one*, vol. 12, no. 5, 2017.
- [39] O. Rudy, M. Garcia-Gasulla, F. Mantovani, A. Santiago, R. Sirvent, and M. Vázquez, "Containers in hpc: A scalability and portability study in production biological simulations," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 567–577.
- [40] L. Benedicic, F. A. Cruz, A. Madonna, and K. Mariotti, "Sarus: Highly scalable docker containers for hpc systems," in *International Conference on High Performance Computing*. Springer, 2019, pp. 46–60.
- [41] Xla: Optimizing compiler for machine learning : Tensorflow. <https://www.tensorflow.org/xla>, last accessed 15. June 2020.
- [42] C. Leary and T. Wang, "Xla: Tensorflow, compiled," *TensorFlow Dev Summit*, 2017.
- [43] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein *et al.*, "Glow: Graph lowering compiler techniques for neural networks," *arXiv preprint arXiv:1805.00907*, 2018.
- [44] S. Cyphers, A. K. Bansal, A. Bhiwandiwala, J. Bobba, M. Brookhart, A. Chakraborty, W. Constable, C. Convey, L. Cook, O. Kanawi *et al.*, "Intel ngraph: An intermediate representation, compiler, and executor for deep learning," *arXiv preprint arXiv:1801.08058*, 2018.
- [45] W.-F. Lin, D.-Y. Tsai, L. Tang, C.-T. Hsieh, C.-Y. Chou, P.-H. Chang, and L. Hsu, "Onnc: A compilation framework connecting onnx to proprietary deep learning accelerators," in *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. IEEE, 2019, pp. 214–218.
- [46] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, "Intel math kernel library," in *High-Performance Computing on the Intel® Xeon Phi*. Springer, 2014, pp. 167–188.
- [47] L. Brown, "Gpu accelerated deep learning with cudnn," *GTC*, 2015.
- [48] Hlrs - high-performance computing center — stuttgart. <https://www.hlrs.de/home/>, last accessed 17. June 2020.

- [49] User namespaces and fakeroot. https://sylabs.io/guides/3.5/admin-guide/user_namespace.html#adding-a-fakeroot-mapping, last accessed 15. June 2020.
- [50] Tensorflow: Build from source. <https://www.tensorflow.org/install/source>, last accessed 10. June 2020.
- [51] Pytorch: From source. <https://github.com/pytorch/pytorch#from-source>, last accessed 10. June 2020.
- [52] Glow. <https://github.com/pytorch/glow>, last accessed 10. June 2020.
- [53] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: <http://yann.lecun.com/exdb/mnist/>
- [54] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.
- [55] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.
- [56] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
- [57] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, no. 3, pp. 211–252, 2015.
- [58] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
- [59] S. Hochreiter, Y. Bengio, P. Frasconi, J. Schmidhuber *et al.*, "Gradient flow in recurrent nets: the difficulty of learning long-term dependencies," 2001.