Available online www.ejaet.com

European Journal of Advances in Engineering and Technology, 2023, 10(9):104-112

Research Article ISSN: 2394 - 658X

Optimizing Code Performance for Machine Learning Models

Kailash Alle

Sr. Software Engineer, Comscore Inc
kailashalle@gmail.com

ABSTRACT

Due to its effectiveness in resolving issues like speech recognition and picture analysis, artificial intelligence (AI) systems based on Deep Neural Networks (DNN) or Deep Learning (DL) have gained popularity. High Performance Computing (HPC) has been a major factor in the development of AI, and training a DNN is a computationally demanding process. Cloud and HPC infrastructure have come together thanks to virtualization and container technology. The challenge of installing and optimizing AI training workloads is increased by these heterogeneous hardware infrastructures. Target-specific libraries, graph compilers, and better data mobility or input/output can all be used to optimize AI training deployments in cloud or HPC environments. By producing code that is optimized for a target hardware or backend, graph compilers seek to maximize the execution of a DNN graph.

The MODAK tool is designed to optimize the deployment of applications on software-defined infrastructures as part of the SODALITE project, which is a Horizon 2020 initiative. MODAK maps ideal application parameters to a target infrastructure and creates an optimized container using performance modeling and data scientist input. This paper reviews container technologies and graph compilers for artificial intelligence, and introduces MODAK. We demonstrate how Singularity containers and graph compilers can be used to optimize AI training deployments. Custom-built, optimized containers perform better than the official DockerHub images, according to evaluation using MNIST-CNN and ResNet50 training workloads. Additionally, we discovered that the target hardware and neural network complexity affect graph compiler performance.

Keywords: Optimizing Code, Performance, Machine Learning Models, Neural Networks

---

INTRODUCTION

Recently, there has been a push for Artificial Intelligence (AI) due to the increasing availability of data and the processing capacity of High-Performance Computing (HPC). In many fields, including speech recognition and visual object recognition, deep learning (DL), a branch of artificial intelligence, has significantly raised the bar. DL uses multiple layers of neural networks to gradually extract higher level features from raw input. References [1], [2].

Convolutional Neural Networks (CNNs) [3] and AlexNet [4] gained popularity in 2012, which sparked this expansion. With the use of GPUs, AlexNet reduces training time and is computationally expensive. DL training workload development and deployment are made easier by frameworks such as TensorFlow [5], PyTorch [6], MXNet [7], and CNTK [8]. The high-level language Keras is supported by certain frameworks as well [9].

The complexity and size of DL training networks have increased recently. Tasks for learning AI are being implemented on cloud and HPC, two examples of heterogeneous hardware targets. The user experience on HPC systems varies greatly from that of cloud settings. Solutions utilizing Subject Specific Standard (DSL), such as Terraform [10] and Cloudify [11], are available in cloud settings to make the management of application models easier. This includes application model deployment, monitoring, and maintenance. To manage the application lifecycle, however, HPC systems need specialized understanding of the system as well as command line tools.

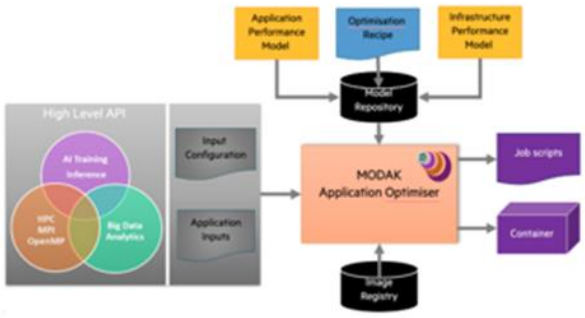
SLURM [12] and TORQUE [13] workload managers are used to submit jobs to computing nodes, and they can be accessed by methods like Secure Shell (SSH). Comparing HPC systems to the cloud for experimentation might present a significant challenge for domain scientists.

The challenge of creating, refining, and implementing AI applications in heterogeneous infrastructures, such as cloud or HPC settings, is multifaceted. The Heterogeneity Alliance's [14] EU projects, such as COLA [15], TANGO [16], HiDALGO [17], Exa2Pro [18], EcoScale [19], and SODALITE, seek to provide the software tools, techniques, and know-how needed for next-generation applications to leverage heterogeneous hardware. By offering tools for software-defined infrastructures, SODALITE [20], an EU Horizon 2020 project, seeks to address the challenge of delivering workloads across heterogeneous environments. MODAK is a model-based application deployment optimizer for static optimization in software-defined infrastructure that we have created for SODALITE. This paper presents MODAK and assesses its application in optimizing AI training workloads through the use of graph compilers and containers.

LINKED PERFORMANCE

HPC users place a high value on application performance and scalability. In order to tailor the application parameters to the target hardware, the optimization procedure typically entails manual profiling. Moreover, because of the variety of hardware in HPC systems, it is not portable and must be repeated when transferring to different HPC systems.

Models that are useful for predicting performance and analyzing how various hardware components impact performance are necessary for automating application optimization on cloud and HPC systems, an increasingly difficult endeavor.



by the large range of hardware options and cloud offerings that are accessible. [21] developed a performance model using application profiling and historical data collected on cloud and HPC platforms. Machine learning, genomics, and molecular dynamics apps were tested on several public clouds using ParaOpt [22], a program that automatically adjusts application configurations for various instance types based on runtime and cost.

The functionality of cloud environments was examined in several studies. In addition to creating a software tool for the ongoing assessment of different cloud environments, Exabyte also used the Linpack benchmark [23] to compare cloud targets.

Utilizing the DiRAC application benchmarks, EPCC directly compared the performance of HPC on-premise systems with the Oracle cloud cluster [25], identifying problems with the scalability and usability of cloud-based clusters.

Numerous instruments are created to enhance the deployment of bundled applications in containers. A Kubernetes container tuning framework is called ConfAdvisor [26]. The AWS Compute Optimizer [27] uses past utilization indicators to optimize workloads for both cost and performance. On the Google Cloud Platform, Google [28] provides optimized containers for the deployment of AI applications. Charliecloud [30] was used in the HPAI project [29] to investigate the viability of implementing AI workloads on HPC systems.

An application optimizer for software-defined infrastructures, such as cloud and HPC, called MODAK will be covered in the next section.

MODAK

Giving apps the flexibility to execute efficiently on a variety of devices reduces costs and saves time in a world where hardware and software rules are abstracted. This is known as a "software defined" environment.

SODALITE [20]

By offering tools for software-defined infrastructures (SDI), which abstract away hardware dependencies and place the computing infrastructure under software control, a European Horizon 2020 project seeks to address the challenge of delivering workloads across heterogeneous environments. As a result, user exploitation of heterogeneous targets, such as edge devices, cloud environments, and HPC clusters, is made easier and better.

The software-defined application installation static optimization is made possible by MODAK, a part of SODALITE. Applications' and the infrastructure's performance models can be used to forecast how well an application will perform when it is deployed in a particular environment.

Running common benchmarks across various deployment infrastructure and application workload settings is how the performance models are created. A linear statistical model is then constructed.

With reference to the target infrastructure's performance metrics, including memory bandwidth and peak performance, MODAK is informed by this model regarding how the application parameters, like the size and format of the input data, impact performance. With this information, MODAK creates an optimized container by mapping the ideal application settings to the infrastructure target.

Model A and its dependencies are depicted in Figure 1. The infrastructure, configuration, and application itself all play a major role in optimizing application performance. For static optimization, MODAK supports three main application types: classical HPC, big data analytics, and AI training and inference. Selecting application optimizations with the SODALITE IDE is done by the data scientist, or AoE. Among the optimizations are modifications to the runtime, environment, or application configuration. For better application performance, additional autotuning of the runtime settings is possible.

The application code, together with the application inputs and configuration, must be defined in a common high-level API for the Application Optimiser to implement the optimizations. As a result, the Optimizer is able to decide how to perform in light of the available target. The Optimizer builds an optimized container for the application deployment by modifying the pre-built, optimized containers from the Image Registry. In order to submit task scripts to HPC schedulers, the application optimizer additionally modifies the runtime and deployment settings. A thorough explanation of each component of MODAK can be found in [31], although it is outside the purview of this work to do so.

HISTORY

For the purpose of optimizing AI training for MODAK, this subsection offers a concise summary of the container solutions, AI frameworks, and graph compilers that are modeling.

Technology Used in Containers

The 1979 release of the Unix chroot command served as the inspiration for the technology known as containers [32]. Using OS-level virtualization, which is significantly less expensive and more scalable than hypervisor virtualization, Linux containers (LXC), an early version of the technologies outlined below, use OS-level virtualization to isolate processes and resources in distinct user namespaces [33, 34].

Virtualization is accomplished by Docker [35] using cgroups for resource control and LXC for kernel-level namespace isolation.

Even though its design presents performance and security risks on HPC computers, it is an industry standard in cloud environments.

Its use of root daemons to generate and execute containers gives users privileged access to the host system's network filesystem, and it specifically does not support multi-user HPC systems.

NERSC created Shifter [36] and LANL developed Charliecloud [30], both of which were more suited to conventional HPC processes because they were designed with HPC systems in mind. Specifically designed to meet the stringent security standards imposed by sites, Charliecloud is a lightweight containerization platform built on a User Defined Software Stack (UDSS). The current lack of community uptake (Charliecloud) and a significant administrative overhead (Shifter) are drawbacks.

A nice middle ground between the cloud standard Docker and the HPC-specific Shifter and Charliecloud appears to be reached by Berkeley National Laboratories' Singularity [37] [38].

In addition to providing native support for resource managers (such as Slurm, Torque, and others), job schedulers, and some MPI features, it was designed with HPC systems in mind. It also provides users with an easy-to-use containerization procedure. Because of its privilege approach, containers are launched as child processes using SUID and non-privileged user namespaces.

With intentions to expand to Docker, we determined that Singularity was the best option for container deployment. [39], [40] provide additional reading on container technology.

Composers of Graphs

The depiction of neural network models as computational graphs using intermediate representations (IR) is a ubiquitous method used by many AI frameworks. Nodes in the graphs indicate tensor operations, while edges show the linkages between the data. High-level IRs are located in the user-facing front-end of the framework, whereas low-level IRs are located in the back-end. Generally, there are several layers of IR.

Optimizing the resulting graph IRs can be done with a collection of framework-specific compilers. Both low-level tensor compilers, which concentrate on building high-performance operators for compute-intensive operations, and deep learning compilers, which concentrate on high-level IR optimizations followed by offloading to vendor-specific libraries, can be used to categorize these graph compilers.

An accelerator for linear algebra designed specifically for TensorFlow is called XLA (Accelerated Linear Algebra) [41] [42]. It takes a graph specified in the High Level Optimiser (HLO) IR and applies target-independent analysis and optimization, including buffer analysis and operation fusion. After the HLO IR is optimized, it is delivered to the back-end for additional hardware-focused HLO-level optimizations. With LLVM, the final code is produced.

By lowering the graph into a two-phase IR, with a heavy focus upon the minimal IR, GLOW [43], tiny for plot lowering, optimizes PyTorch models. Machine-specific code is generated at the lowest level, while the high-level IR is used for specific to a domain optimizations and the low-level instruction-based, address-only IR is used for memory-related optimizations.

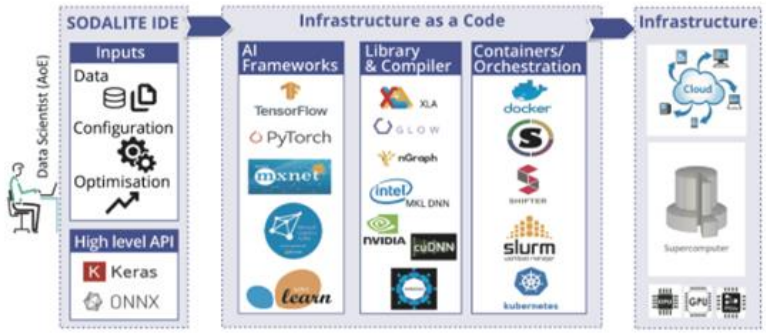
Many other frameworks, including TensorFlow and PyTorch, can utilize nGraph [44], a framework-independent graph compiler. In order to produce code that is optimized for a particular back-end using vendor-specific libraries, it serves as a bridge, translating the graph model unique to the framework to a common, intermediate high-level IR.

Approach

Here, we show how to use containers and graph compilers to optimize AI training workloads using MODAK.

Illustration of MODAK Artificial Instruction

An example of using MODAK for AI training application deployment is shown in Figure 2. With input data, configuration, and optimizations, a data scientist builds the application model using the SODALITE IDE. The program is then deployed in a Docker or Singularity container with the chosen AI framework and optimized libraries like MKL, having been built in a high-level language or API like Keras or ONNX [45].



. Compilers such as XLA and GLOW, as well as cuDNN [47] or [46].

Then, an HPC or cloud system receives this optimized, containerized application.

The data scientist inputs optimization parameters into MODAK by encoding them in a JSON format within the IDE. Listing 1 displays a portion of a TensorFlow deployment's optimization DSL.

This is where you enable the XLA compiler and special build optimizations for a chosen target (x86 and NVIDIA). TensorFlow containers are prebuilt by MODAK, and tags are applied based on available optimizations. MODAK chooses the optimized container based on the DSL optimizations that have been chosen. During deployment, MODAK is also capable of building a container.

```
"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 }}}
```

Modules for Artificial Intelligence Architectures:

Based on the SODALITE HPC testbed at HLRS, the University of Stuttgart's research and supercomputing center, we developed and assessed AI framework containers for MODAK [48]. An Intel(R) Xeon(R) CPU E5-2630 v4 processor, five computing nodes, an Nvidia GeForce GTX 1080 Ti GPU, and 125GB of main memory make up the testbed. The front-end node, which runs Torque, is the center of attention. Access via https is available on all nodes. and afternoon 2 p.m. to 5 p.m. and can even set intervals of each appointment like 30 mins.

• Patients must register to PASS by providing necessary information related to their insurance and address. Once recorded they can login into PASS and search nearby doctors who are using PASS and by selecting any doctor, a patient can be able to see that doctor chosen's availability like what all time slots on what all dates the chosen doctor is available.

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	

We used factors like popularity, image availability, build instructions and documentation clarity to decide which frameworks and graph compilers to benchmark. Table I provides a list of the graph compilers, versions, pictures, and sources for the AI framework. Project websites provided source build instructions (opt-build), or official project images could be obtained from DockerHub (Hub) and packaged as Singularity containers using the Python package manager (pip).

TensorFlow XLA, for example, and other optimization libraries were made sure to be compatible with both the official and our images. The TensorFlow framework auto-builds XLA, despite its separate listing in the table. Specific versions of TensorFlow are supported for XLA and nGraph.

With Singularity pull, Docker containers are directly ported to Singularity.sif files, making it easy to obtain the DockerHub containers. As a starting point for comparison, we selected the photos that were labeled with the necessary version and a CPU or GPU target.

We prepared definition files for Singularity in order to build the remaining containers. Several sections for pre-build preparation, file importation into the container, container environment setup, post-OS installation container commands, etc. are included in the definition files, along with a header that identifies the operating system (OS) used within the container.

Our custom-built containers utilize two base OS containers, which are called upon in the specification header, because the dependencies between them vary based on whether the containers run CPU or GPU workloads.

CPU-Related Structures

As the foundation OS in the header, the CPU-enabled containers use a specially created Ubuntu 18.04 image. Python3, clang-8, and llvm-8 packages are included in the image. The post-OS installation section contains the remaining build instructions. In the case of pip-based containers, this entails adding commands to install extra dependencies and then using pip to install the framework or compiler in accordance with project guidelines.

Comparably, all installation instructions for the source build containers are encoded in the post section and follow the guidelines provided by the documentation for each specific project [50]–[52]. In order to enhance CPU speed, compiler optimization flags were set where appropriate. Specifically, TensorFlow employs the build tool Bazel, which takes compiler options as input via the parameter.

GPU Modules

The underlying operating system for the GPU containers is Ubuntu 18.04, together with NVIDIA DockerHub images that include the NVIDIA-kernel, cuda toolkit 10.1, and cudNN7. Since retrieving cudNN7 via the command line is not possible, we opted for the NVIDIA base image to minimize portability concerns and facilitate distribution.

Next, to enable their use for source builds, all NVIDIA package paths are established in the container environment section.

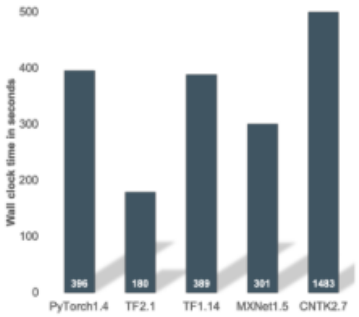
The remaining contents of the container construction file resemble the CPU containers' contents. The source build commands have been adjusted in the file's post section to account for variations in GPU builds, particularly with regard to the TensorFlow build.

Next, the Singularity build command is used to generate both CPU and GPU containers, with the --fakeroot flag enabled.

This build may take a few minutes to several hours, depending on the framework or graph compiler. The created containers can then be launched to carry out particular tasks or executed interactively. Be aware that GPU containers are subject to severe limitations imposed by Singularity: the container's nvidia-kernel version must match the host's nvidia-kernel version, along with any additional requirements like CUDA. Using the Singularity NVIDIA flag --nv when starting containers will get around this restriction.

Comparisons

Through the use of image classification training and tracking the completion of a predetermined number of epochs, we were able to quantify container performance. We used MNIST [53] for the CPU workload and ImageNet [54] for the GPU task during training in order to accurately evaluate the frameworks on both CPU and GPU. The MNIST training problem is an image classification task for handwritten numbers [55] [56]. The term "MNIST" itself describes a dataset of 60,000 grayscale pictures with handwritten numbers ranging from 0 to 9.



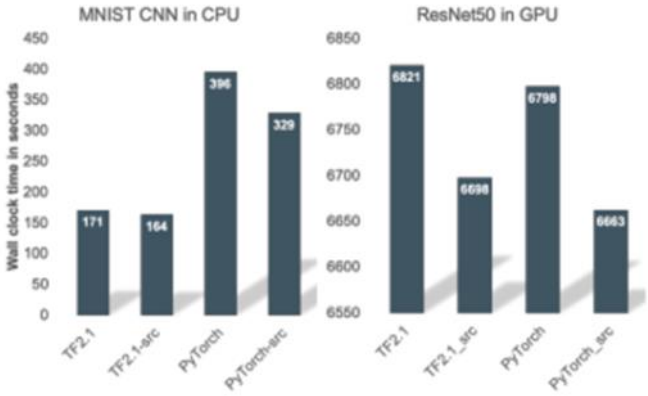
Scheduling and managing appointments from Scheduler module

ResNet50 [58] residual network, a fifty-layer deep neural network, was the network utilized for ImageNet training. The issue of vanishing or bursting gradients is minimized and deeper networks can be used with residual networks by using skip connections to pass residual functions [59]. Single precision, 96-batch size, and three epochs of training were used to all ResNet50 benchmarks

OUTCOMES

We address preliminary findings from our benchmarks in this section. In every picture, the Y-axis represents the average time per epoch of ResNet50 training or the wallclock time in seconds needed to finish 12 epochs of MNIST training.

Additionally, src post-fix refers to the best source created container, TFx.x refers to TensorFlow version x.x, and XLA or NGRAPH post-fix indicates that the graph compiler is enabled. For all comparisons, PyTorch version 1.4 is utilized.



The DockerHub containers' performance for MNIST CNN training on CPU alone is displayed in Figure 3's results.

For these outcomes, graph compilers are not activated. As can be seen, TensorFlow 1.14, PyTorch, and MXNet (v1.6) all behave similarly, with the exception of TensorFlow 2.1, which performs nearly 54% better than TensorFlow 1.14. This is probably because TensorFlow 2.0 enabled eager execution by default, whereas TensorFlow 1.14 uses graph execution. The official documentation mentions that CNTK (v2.7) is a far outlier because it lacks CPU optimizations.

Please take note that the evaluation of MXNet and CNTK was limited to comparison; no other containers were assessed other than those obtained from DockerHub.

The training times of MNIST CNN in custom-built and DockerHub containers are contrasted in Figure 4 (left). The TensorFlow custom container outperforms the official DockerHub container by only 4%, but the PyTorch container outperforms the official one by a significant 17%.

With a specially designed AI framework for the NVIDIA GPUs, Figure 4(right) displays the outcome of ResNet50 training on ImageNet data. For TensorFlow and PyTorch source-built containers, there is a noticeable 2% increase in performance. For MXNet containers, we observe comparable performance.

CONCLUSION

Workloads for AI training can be distributed among heterogeneous targets with flexibility thanks to software-defined infrastructures. Using a range of hardware, working with various setups and data types to optimize AI workloads is essential. Using performance modeling and container technology, we present MODAK, a new tool that maps ideal application characteristics to infrastructure. Graph compilers and Singularity containers were utilized to optimize AI training deployments, demonstrating the application of MODAK. Using specially designed,

optimized containers, we demonstrated a speedup of improvement to 17%, and using graph compilers, we demonstrated a speedup of up to 30%. Graph compilers were also shown to slow down training in some usage cases. Afterwards, a performance model and comparable training workload optimization are applied using these benchmark findings.

UPCOMING PROJECTS

To benchmark GLOW and Graph on PyTorch, MXNet, and any upcoming compilers, more work is needed. We intend to expand the capabilities of our containers to include other HPC-specific workloads, such MPI applications. Lastly, even though Singularity containers are the recommended runtime for HPC, we will convert the images to Docker to facilitate project distribution.

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 largescale 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'Alche-Buc, ´ E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.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. Klusa´cek, V. Chlumsk ˇ y, and H. Rudov ` a, "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. Bazhirov, "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, "Confadvisor: 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 devloperWorks 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. Rudyy, M. Garcia-Gasulla, F. Mantovani, A. Santiago, R. Sirvent, and M. Vazquez, "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. Bhiwandiwalla, 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 System (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