HiT: A framework for increasing portability of deep learning applications in HPC

Vineeth Gutta

May 2021

A technical report submitted to the Department of Computer and Information Sciences at the University of Delaware in partial fulfillment of the Preliminary research requirements

Prelim Committee:

Advisor: Dr. Sunita Chandrasekaran

Dr. Chandra Kambhamettu

Abstract

The proliferation of Deep Learning (DL) models in recent years fueled a growth in the number of open source DL frameworks. Once a user chooses a framework there can often be constraints on the hardware accelerators that can be used to train or run models with. Even worse, the choice of their framework can hamper their ability to choose the best hardware accelerators for training and inference. In this research project we introduce Hardware independent Translator (HiT), a toolchain that aims to increase portability of DL based applications across different kinds of systems. Eventually the goal for HiT is to provide this portability across various hardware architectures. This includes support for application specific integrated circuits (ASICs) such as Tensor Processing Units (TPUs) that can give users the ability to use the vast compute capability some of them have to offer.

The toolchain, HiT, will achieve its goal of increasing application portability by incorporating a variety of tools and techniques. The toolchain will include a high-level abstraction for creating portable models, an Intermediate Representation (IR) layer that translates model graphs from one framework to another and containerization across systems. Other features of the tool chain could also include support for a growing list of DL model operators such as dropout. The toolchain will be evaluated using an MLPerf like benchmark suite among other case studies.

This report demonstrates a preliminary research of the potential high-level abstraction, followed by evaluating a workflow that entails an IR targeting new hardware runtimes and explores the benefit of containerization of models across systems that we have access to.

1 Introduction

The number of frameworks available for deep learning is extensive and there is no one framework that dominates in usage [16]. Some examples of these frameworks include PyTorch, TensorFlow, and Scikit-learn. Each of them have their strengths and weaknesses

but for the most part the differences are minimal, where most algorithms can be written in any of them, and the choice of framework comes down to individual or team preferences. Some of these frameworks have back-ends that are optimized for certain hardware accelerators as training a model can be a very compute intensive task. This often means that users are locked into not only the software but also the hardware accelerators they can use to train their models. Most users also do not own the hardware accelerators they use for training. That means they rely on high performance computing (HPC) systems like UD's DARWIN [7] cluster to share computational resources with other users of these large systems. DARWIN, like other similar systems, has many hardware accelerators available for its users. Systems made by vendors such as NVIDIA and AMD have slightly different GPU architectures. In addition, there are architectures that are being designed and built specifically for deep learning type workflows and are broadly known as application specific integrated circuits (ASICs). Google's Tensor Processing Unit (TPU) is an example of an ASIC and it is designed to work with TensorFlow. When accessing shared compute resources, users may not get access to the systems they request leading to long wait times. So there is a need for a tool that can allow users to seamlessly transition between different types of systems without the need to manually change the model or the code to run on a new target architecture.

In this research, we introduce Hardware independent Translator (HiT) to solve compatibility issues between the supported architectures and high level DL frameworks to allow applications to run on a target architecture. ONNX [10] translates models at the IR level of a compiler, which translates the high level code to the machine level to represent these models, which are typically DL, and consist of a set of operators which are translated into an intermediary format also known as ONNX. ONNX format integrates well with ONNX Runtime [10] because the latter supports various types of hardware accelerators in its backend. ONNX Runtime allows HiT to maintain performance, which is critical during training and inference of deep learning models, and in some cases manages to improve the performance on a target architecture compared to the original one.

The main contributions of this research are:

- A software prototype, HiT, that demonstrates portability of DL applications using a combination of container environments and graph/IR based DL model translators to demonstrate the ability to target these applications on various hardware runtimes and show the performance impacts of doing so.

- A new framework to create a more advanced IR level conversion wrapper based on what we learned through the research done for the previous contribution

In the rest of this report, we go into more details regarding various aspect of this research such as the motivation and give the background details about some crucial parts of this research before presenting the framework and the results of our experiments.

2 Motivation

There are many HPC+AI workflows that are designed to run on specific systems based primarily on the hardware available at the time of development. But hardware architectures evolve and organizations acquire different kinds of systems based on factors such as cost, efficiency and performance. And often these decisions are made for the benefit of the wider scientific community. As the Office of Science reports [6], there are substantial

needs for extreme-scale computing and the needs vary widely across a growing range of workloads. The report also states the growing challenges of making scientific applications more portable as the complexity and diversity of HPC grows. As cutting edge HPC transitions into exascale level computing, hardware vendors that can maximize performance of these systems are chosen.

At the Department of Energy's (DOE) Oakridge Leadership Computing Facility (OLCF), applications running on the current Summit supercomputer [29] need to eventually be transitioned to the upcoming Frontier supercomputer [14]. In this case, the transition will be from NVIDIA GPUs to AMD GPUs. This has several implications including the necessity to port applications and models from NVIDIA's CUDA runtime to AMD's runtimes such as MiGraphX. For developing software to run on Frontier, many traditional low level programming languages like C, C++, and Fortran are supported high level options like python will have limited GPU support according to OLCF [25]. Most DL framework users write in python so it will be challenging for these users to transition their DL workflow from NVIDIA and CUDA environment to an en environment with a different hardware architecture such as AMD's GPUs.

Another example is an Exascale Compute Program (ECP) project called CANDLE (CANcer Distributed Learning Environment) [31, 32]. The project's goal is to address three separate challenges related to cancer surveillance with pilot applications. CANDLE mainly uses python across a few various DL frameworks. Here again there is need for increased portability of CANDLE. As mentioned earlier in the Frontier example, applications written in high level languages like python with frameworks make it challenging to port to a system with a different hardware architecture.

This problem is not limited to large scientific applications that require extreme-scale computing. HPC clusters such a DARWIN offer a combination of accelerators including NVIDIA and AMD GPUs. If researchers wanted to run their applications on a different system, including due to availability of resources, the process is not simple even for small workloads. There is a need for a high level abstraction that can allow users to port their applications between systems that offer different hardware accelerator architectures much more seamlessly compared to the process it currently requires to make than transition.

3 Background

To our knowledge, our research is the first to address incompatibilities between DL frameworks, models, and hardware architectures. In this section, we describe the work the HiT builds on. Numba [20] is the most related in terms of its objective. It is a just-in-time compiler for numerical functions in Python that has support for certain types of hardware acceleration including CPU multi-threading and GPUs from NVIDIA and AMD. The libraries that Numba supports, such as NumPy [30], are widely used attracting many users. But NumPy, which is used mainly for numerical computing, does not offer the functionality to build complex ML models unlike DL frameworks. In addition, the main goal of Numba is to offer parallelism whereas HiT's goal is to remove the hardware constraints when accelerating models during training and inference.

HiT leverages several software libraries to provide its functionalities. For example, ONNX provides IR (Intermediate representation) level translation of models between various popular frameworks such as TensorFlow and PyTorch. In addition, ONNX Runtime allows models in ONNX format access to hardware level acceleration from various

providers such as CUDA, TensorRT, and MiGraphX. In addition to these libraries, we also use MLPerf [27], a benchmark suite, as a reference since it provides results for training and inference of leading DL models on various hardware accelerators such as NVIDIA's V100's, Google's TPUs, and the A64FX which is an ARM based accelerator currently being used in Ookami which is the world's faster supercomputer.

The rest of this section will go into details on the background of each of the major libraries used and how they fit into the overall context of this research.

3.1 Frameworks

In recent years, machine learning became mainstream and gained popularity because of the advances made using deep learning that enabled various tasks such as speech recognition, object detection, and many other fundamental problems to be tackled with much greater accuracy. This was mainly driven by an increase in the availability of data and compute resources. Increased usage of ML and DL models necessitated frameworks to help users build these machine learning models for not just research but also for software development. These frameworks are high level and productivity-focused with the low level performance-oriented interface with hardware mostly abstracted from the users. These frameworks are also rapidly evolving based on the needs of their users. And because most of them are open sourced their most ardent users often add features to these frameworks. One such example framework is Scikit-learn [24], which integrates many state-of-the-art (SOTA) machine learning algorithms for medium-scale supervised and unsupervised problems. It also uses several other libraries such as Numpy [30] and Scipy [22] which allow Scikit-learn to store data efficiently in data structure and provide linear algebra functions essential for machine learning. The remaining frameworks discussed in this section in addition to allowing users to build machine learning models also allow for large-scale deep learning to express a variety of algorithms. Some examples of these DL frameworks include TensorFlow [1], Torch [5], Keras [8], PyTorch [23],and CNTK [28].

Most of the DL frameworks adopt a layered software architecture that is similar and provide APIs that enable users to configure DL models and training methods including optimizers. According to [33], these frameworks are implemented on top of various parallel computing libraries: BLAS (Basic Linear Algebra Subprograms) libaries, such as OpenBLAS, cuBLAS, NCCL (NVIDIA Collective Communications Library), OpenMP, and Eigen.

A. TensorFlow and Keras

TensorFlow [1] was originally developed by the Google Brain team for internal Google use to conduct ML and DL research. It is now open sourced and used for both research and production. It is one of the most popular frameworks according to various metrics including search volume, arXiv articles, and GitHub activity [16]. In addition to these reasons, the main focus of the research is framework agnostic for the most part so it was important to test with various frameworks that are supported well and have a large community. TensorFlow was also extensively used in the CANDLE benchmarks further reinforcing the need to pick TensorFlow as one of the frameworks tested. TensorFlow has a dataflow model where tensors flow between nodes along the edges [33]. The nodes represent a mathematical operation while the edges represent dataflow giving users the flexibility to represent complex neural networks.

Keras [8] is a deep learning API that runs on top of TensorFlow and is written in Python. It was developed to enable faster experimentation. It provides essential abstractions for prototyping with much higher velocity. Like TensorFlow, Keras also provides researchers and developers the ability to take advantage of TPUs and GPUs for faster computation. In addition, it also allows models to be exported as graphs to other runtimes such as browsers, mobile, and embedded systems.

B. PyTorch

PyTorch [23], like TensorFlow, is another open source deep learning library originally developed Facebook's AI Research lab. Most of the popular graph-based DL frameworks construct a static dataflow graph to represent the computation which can then be applied repeatedly to batches of data. Although this approach provides visibility into the whole computation ahead of time, it comes at the cost of ease of use, ease of debugging the graphs, and flexibility when it comes to the types of computation that can be represented. PyTorch offered users dynamic eager execution with automatic differentiation mostly without sacrificing performance where operations are evaluated immediately without building graphs [23]. TensorFlow subsequently adopted eager execution and to allow its users similar functionality as PyTorch's dynamic eager execution. In addition to its design principles of putting researchers first and ease of use, PyTorch also offer performance focused implementation using an efficient C++ core, having seperate control and data flow, multiprocessing, and through a custom caching tensor allocator. The combination of these features allow PyTorch to use GPUs with CUDA APIs to saturate the GPU and reach peak performance even though Python has a high overhead because it is an interpreted language.

3.2 ONNX

ONNX (Open Neural Network Exchange)[10] allows conversion of DL models using graphs generated by frameworks discussed in the previous section. These graphs are also colloquially known as the IR (Intermediate representation) layer. ONNX defines a common set of operators and a common file format for using models with a variety of frameworks, tools, runtimes, and compilers. IR is the level where DL models are broken down into operators which are the building blocks of these models. But this is not always possible because of the rapid pace of innovation in DL models where ONNX may not support the latest operators. But ONNX allows users to create custom operators to support the latest DL models in ONNX format. ONNX also interfaces with ONNX Runtime to allow models in ONNX format to infer (and train in the future) which requires interfacing with the hardware runtimes such as CUDA and TensorRT from NVIDIA and MiGraphX from AMD.

3.3 ONNX Runtime

ONNX Runtime [11] supports a wide range of hardware runtimes and by extension many architectures. Some of these include the CPU, CUDA and TensorRT for NVIDIA GPUs, MiGraphX for AMD GPUs. Support for the latter is limited as MiGraphX integration is in beta. The combination of ONNX [10] and ONNX Runtime will allow us to create prototypes of DL model pipelines that demonstrate the increased flexibility in hardware

architecture choices this software combination allows. In addition to that, experiments, including our own, have shown that ONNX Runtime can offer performance speed up at inference time in certain DL models that are optimized for ONNX Runtime.

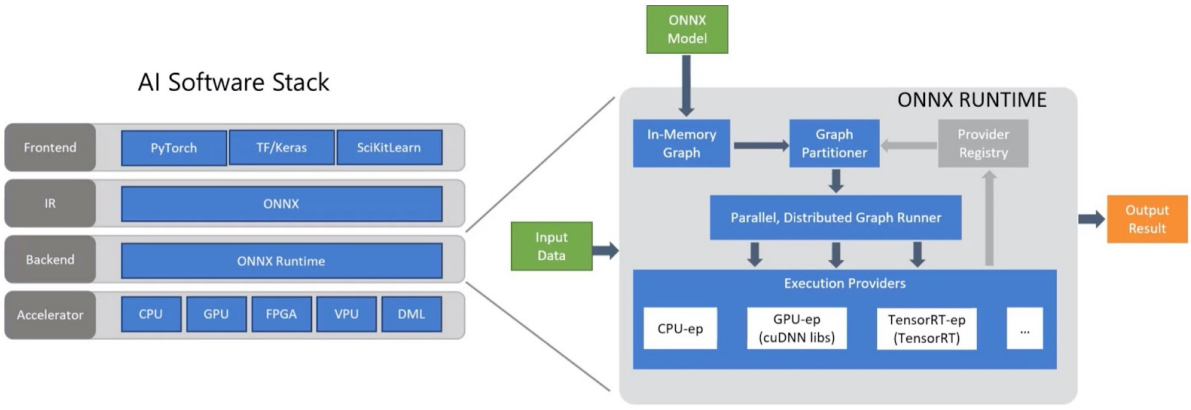

Figure 1: Visualization of the ONNX ONNX Runtime Stack with respect to high-level frameworks and low-level accelerators [15].

3.4 TVM

For further performance optimization of DL models, TVM (Tensor Virtual Machine) [4] can help optimize DL models in ONNX format at the IR and compiler level. It's main objective is to help bridge the gap between high level productivity-focused software frameworks and low level performance and efficiency-oriented hardware backends. TVM uses several techniques including operator fusion which combines multiple operators into a single kernel without saving the intermediate results in memory. This optimization can be particularly useful at inference time. TVM also contains an *automated schedule optimizer* that can find optimal operator implementations for each layer of a DL model. But often the search space to choose schedule optimizations is large for each hardware back-end. To solve that it uses ML-based cost model and design choices to optimize finding best schedules. The combination of ONNX, ONNX Runtime, and TVM can potentially help improve performance of DL models.

3.5 Containerization

One of the benefits of containerization is being able to install the benchmark, dependencies and its corresponding environment variables in a container image and using the image to run in a number of different clusters rather than installing each of these components independently on every system of interest. Containerization would also avoid any conflict with libraries that may be installed by the system admin or other developers. It will also allow a seamless usage of different DL frameworks that may typically come with conflicting software dependencies while running on the same system. We will also be able to allocate relevant compute resources in a way that we could run multiple instances of a DL framework concurrently.

Containerization also comes handy even if the underlying platform is from the same vendor, such as NVIDIA's GPUs, migrating the code to a new runtime, for example from CUDA to TensorRT, which can both run on the same NVIDIA hardware will still

require a different set of software dependencies. The problem is even more pronounced when switching hardware from say NVIDIA GPUs to AMD GPUs. Containerization makes this process more streamlined because the containers can be prebuilt with the prerequisite software dependencies to run on a different runtime environment. This is especially true for compute clusters such as UD's DARWIN because installation of niche software can be restricted and the process to do so can often be cumbersome if it is not officially supported by a HPC compute cluster. Containers alleviate this bottleneck because they provide secure and isolated virtual environments that can come with the required dependencies. Because of all these advantages, migrating the research workflow to work in containerized environments will be an important aspect of this work. There will be some prerequisites for containerizing including support from the compute clusters or systems for container software such as Docker [21], Singularity [19], or Kubernetes.

A. Docker

Docker is an industry standard container software that promises the ability to package applications and their dependencies into lightweight containers that can start up quickly, create an isolated environment, and can be moved easily between different systems [21]. But Docker creates security issues and causes other problems in HPC environments. In terms of security, Dicker containers have privileged access to the host systems network file system making it unsuitable. In addition, Docker uses cgroups to isolate containers and that conflicts with the Slurm scheduler used by many HPC systems for resource allocation [2].

B. Singularity

Even though Docker creates security concerns and is incompatible with HPC systems, its most important feature of isolated environments and lightweight nature can still be useful in HPC context. Singularity [19] is an alternative to Docker for containerization in HPC environments. It provides reproducible, portable, shareable, and distributable containers just like Docker but at the same time it assumes a "no trust" security model where untrusted users run unstrusted containers. In addition, Singularity supports HPC hardware and scientific applications because of its support for MPI. Singularity also allows users to bring external driver modules from the underlying system into the container environment and because of this it has built it options to make GPUs, from both NVIDIA and AMD, available to the containers.

Studies in [2] show that it is possible to deploy DL frameworks at scale on HPC systems using containers. But the major distinction between their work and ours is that they did not use GPUs while we use them in addition to CPUs. Furthermore, they use Charliecloud [26] for their containerization software whereas we use both Docker and Singularity depending on the system we run our experiments on. This is because of the drawbacks of Docker in HPC environments mentioned previously requring us to use Singularity as the alternate container software on HPC clusters.

4 Prelims Research

As mentioned previously in the background section, portability of DL based scientific applications like CANDLE is an important challenge. We create a workflow, named HiT, using a combination of containerization, ONNX, and ONNX Runtime to show that portability is possible. In addition to achieving portability, our experiments show that this workflow does not have a large overhead in terms of latency of DL models at inference time in ONNX Runtime using ONNX format compared to inference in the native framework. And this was tested across both CPU and various GPUs.

ONNX provides a suite of tools to assist users when converting their models from various DL frameworks to ONNX format. Some frameworks such as PyTorch have that functionality built into the framework natively while ONNX also provides external (non-native) libraries that can be used to convert from other frameworks that do not have ONNX conversion tools natively. Some of the frameworks where this is true include Scikit-learn, TensorFlow and Keras. For each of them, ONNX has built separate libraries such as sklearn-onnx, tensorflow-onnx, and keras-onnx respectively to convert to ONNX format. Even with all these libraries conversion of models must be done properly for inference to work in ONNX Runtime. In addition to that, the helper conversion libraries available to use from the DL frameworks convert a single NN graph and are not reliable when a user converts an ensemble of models. To solve this, we propose an ONNX conversion wrapper that can allows users to convert an ensemble of DL models to the ONNX format for inference in ONNX Runtime. Based on our experience from this research so far, a program that offers this functionality needs to have a few crucial components to ensure that an ensemble of models are converted accurately into ONNX format.

The following is high-level abstraction created based on the lessons learned during this research

1. Take the following from user as input
    - DL framework of origin
    - The trained models that are part of their ensemble
    - Each model's dummy input where each of the models can ingest this data to execute each of the models once to construct their graph
    - Some training/testing data for each of the models to test equality after conversion
    - Hardware runtime/architecture user will using for inference
2. Import all necessary libraries for conversion (based on DL framework of origin)
3. Convert each of the models into ONNX format
    - ONNX helper methods convert with Tracing methodology using user provided dummy input
4. More robust validation. User must have option to override this for models that behave stochastically

- After conversion is successful for each model, where ONNX validates the model, we execute the converted model in ONNX format (run inference) once with the additional train/testing data user provided.
- Then we run inference on the original model
- Complete if outputs of both models are same

5. Optimize operators of the converted models with TVM [4]

   - Use TVM's operators fusion feature that transform the computational graph into a fused version that can give improve performance by reducing memory accesses.
   - Incorporate TVM's automated schedule optimizer.

6. Return the ensemble of models in ONNX format.

This abstraction not only streamlines the process of converting a complex ensemble of models into ONNX format but also incorporates a method of improving the performance of DL models at inference time.

4.1 Hardware

Thanks to my advisor, Dr. Sunita Chandrasekaran, I was able to obtain access to a several compute systems with a a broad set of hardware accelerators to run my experiments on. This section will give details about each of those systems.

A. Computational Research and Programming Lab (CRPL) Systems

The first, and probably the most important, system used during the course of this research was CRPL's Leia system at the University of Delaware. It consists of an AMD ThreadRipper 1950X CPU with 16 cores, 32 threads, and 48GB of RAM. In addition, Leia also has 1 NVIDIA GTX 1080 with 8GB GDDR5X RAM with NVIDIA's Pascal architecture and 2 NVIDIA Tesla P40 with 24GB of GDDR5X RAM and also on the Pascal architecture. It was initially running Ubuntu 18.04 operating system but was upgraded to Ubuntu 20.04 during the course of this research. As stated in the next section, all experiments that involved using a GPU on Leia used a single NVIDIA P40 GPU. The main softwares required were Docker and NVIDIA Container Toolkit [9] which was necessary for the Docker containers to have the ability to access the GPU on the host system (Leia). Leia was often the cornerstone of this research because the initial setup and trial runs for many experiments done on the HPC systems that follow in this section were done on Leia. For example, the compute clusters required elevated privileges to build containers using the manifest file such as a Dockerfile. Often this a solved by building the container on Leia and then moving the container to the target system.

The second CRPL lab system used to conduct experiments is Skywalker which consists of the AMD Threadripper 1950X CPU with the same configurations as Leia but it has different GPUs. It consists of 1 NVIDIA Titan V with 12GB of HBM2 (high bandwith memory) and 2 NVIDIA Tesla V100s with 16GB of HBM2. Both have NVIDIA's Volta architecture compared to the older Pascal architecture on Leia's P40s. Like Leia, Skywalker was also updated to Ubuntu 20.04 OS from Ubuntu 18.04. And it also required

Docker and NVIDIA Container toolkit to give the containers access to the GPUs.

B. UDEL's Caviness

Caviness is University of Delaware's third HPC cluster. It consists of 126 compute nodes with a total of 4536 cores and 24.6 TB of memory [3]. Each of the nodes consists of Intel "Broadwell" 18-core processors in a dual-socket configuration for a total of 36 cores per node. Caviness also consists of nodes with NVIDIA P100 GPUs that have NVIDIA's Pascal architecture. Caviness was mainly used to run experiments on this GPU.

C. UDEL's DARWIN

DARWIN (Delaware Advanced Research Workforce and Innovation Network) is the most recent HPC clusters that was commissioned at UD. It consists of 105 compute nodes with a total of 6672 cores, 22 GPUs, 100 TB of memory, and 1.2PB of disk storage. Unlike Caviness, DARWIN has combination of GPUs from both NVIDIA and AMD. It has 9 nodes each a with an NVIDIA T4 GPU consisting 2,560 CUDA and 320 Tensor cores per node. DARWIN also has 3 nodes each with 4 NVIDIA V100 GPUs consisting of 20,480 CUDA cores and 2,560 Tensor cores per node [7]. In addition to the NVIDIA GPUs, DARWIN also has an AMD MI50 GPU. The AMD GPU is of particular interest for this research. Most of the work pertaining to this research done on DARWIN was during its phase 1 early access because it is still in the process of entering full production.

Table 1: Systems Matrix

Systems	**DARWIN**	**Caviness**	**CRPL**
CPU	Intel + AMD	Intel E5-2695	AMD Threadripper
GPU	NVIDIA V100 + NVIDIA T4 + AMD MI50	NVIDIA P100	NVIDIA P40 + NVIDIA V100

Table 2: Specs of GPUs used during this research

GPU	NVIDIA P40	NVIDIA P100	NVIDIA V100	AMD MI50
Peak Single Precision (FP32) Performance	11.7 TFLOPs	9.5 TFLOPs	14.1 TFLOPs	13.3 TFLOPs
Memory Size	24 GB	16 GB	16 GB	32 GB
Memory Type	GDDR5x	HBM2	HBM2	HBM2
Memory Bandwidth	480 GB/s	732 GB/s	900 GB/s	1024 GB/s

4.2 Experimental Setup

The experimental setup for this research consists of variables such as DL frameworks, models, hardware accelerators, and hardware runtimes. Some of these have been discussed in detail in the previous sections. The ones that have not will be explained in this

section. All the models outline below are pre-trained in one of the frameworks outlined in the frameworks section above. This section will include details about the various models chosen for experimentation in this research and some of the reasoning behind their choosing them.

A. AlexNet

AlexNet is a convolutional neural network (CNN) that was originally introduced in [18] and won the ImageNet Large Scale Visual Recognition Challenge in 2012. We chose this model because of the extensive documentation available for it in various DL frameworks. This was important to eliminate problems in framework of origin as the main tests would be performed in a target framework such as ONNX.

B. SqueezeNet

For the second model, we chose Squeeze [17] because of its fundamental similarity to AlexNet in terms of the general problem. Because of that the data at inference time would also be similar. But the most important reason for choosing SqueezeNet was to see the impact of model size on performance after conversion to ONNX. SqueezeNet has 50x fewer parameters compared to AlexNet and the model size is thus smaller in a similar proportion. SqueezeNet manages to achieve this while maintaining accuracy AlexNet achieves [17].

C. BERT

In the past few years, pre-training of language models has significantly increased the size of models. These models deliver state-of-the-art performance across a wide range of NLP tasks. In our experiments, we wanted to incorporate a large language model to compare performance between the native framework and the combination of ONNX and ONNX Runtime across hardware runtimes. We experimented with BERT because of its versatility and performance across eleven different natural language processing (NLP) tasks [12]. BERT has many variations that are fine-tuned for various tasks. For the purpose of this research, we chose the BERT-base-uncased model. Base refers to the original model architecture outlined in [12] and uncased refers to the language model's capability in terms of the case of text it can take as input. This means all input sequences are lowercase.

D. CANDLE

CANDLE (CANcer Distributed Learning Environment) [31, 32], a DOE and National Cancer Institute (NCI) partnership project, consists of three key challenges. The first one, called the "drug response problem," aims to develop predictive models for drug response to optimize pre-clinical drug screening and drive precision medicine-based treatments for cancer patients. The goal of the second challenge, called the "RAS pathway problem," is to understand the molecular basis of key protein interaction in the RAS/RAF pathway that is present in 30% of cancers. The third challenge, called the "treatment strategy problem," aims to automate the analysis and extraction of information from millions of cancer patient records to determine optimal cancer treatment strategies across a range of patient lifestyles, environmental exposures, cancer types, and healthcare systems. The CANDLE challenge problem's goal is to solve large-scale ML problems for the three

applications mentioned above [13].

4.3 Experimental Results

This section highlights some of our preliminary results using the parts of HiT framework that is ready as of now:

Table 3: Models and their inference latencies across CPUs and GPUs

Models	Frameworks	CPU[1]	GPU (P40)	GPU (V100)	GPU (AMD MI50)
AlexNet	PyTorch	2.3s	0.12s	0.07s	
	ONNX Runtime	3.14s	0.17s	0.1s	
SqueezeNet	PyTorch	5.58s	0.21s	0.22s	
	ONNX Runtime	1.29s	0.36s	0.11s	
BERT	PyTorch	3.1s	0.825s	0.9s	
	ONNX Runtime	2.4s	0.3s	0.178s	

The main takeaway from this table is the increased portability achieved by HiT. All the results in table 3 above were obtained inside a container where the container had access to each of the hardware accelerators specified in the table in the corresponding system each of the containers was operating in. See table 1 for more information on what system each of the accelerators were located in and table 2 for their specifications.

Initially, we used Docker on the CRPL lab systems to circumvent the need for elevated privileges to install various software libraries. In order to achieve this, we still required a system admin with elevated privileges to install Docker and Docker Container toolkit [9] on the systems we were using containers with GPUs on. The benefits of containerization soon became apparent as we tried to request access to HPC systems and were preparing to run experiments on them once we gained access. That said, deploying containers in HPC environment brings its own set of challenges. It was important to allocate compute resources, especially the GPUs, in interactive mode on HPC systems for debugging purposes. Submitting jobs in batch mode is less efficient from a user perspective when ensuring proper functionality.

For all the models converted from a DL framework to ONNX, we insured accuracy of the final output during inference done on ONNX Runtime by comparing the output predictions to the predictions made by the DL framework where the model was converted from. In addition, the process of converting models properly with the input and output names set to ensure proper inference after conversion. ONNX also has many opsets because it is a library that is evolving and thus requires the developers of ONNX to update and add support for new mathematical operators necessary for DL algorithms. ONNX also provides functions that help users verify the validity of the ONNX graph generated after conversion. ONNX's built in validation code does this but only with the dummy input the user passes in before conversion to ONNX. It's important to note that this only helps confirm whether the output model graph has a valid schema. ONNX . That is why it is necessary for users of ONNX conversion tools to device ways to verify output model. We did this by passing same input data to the model in ONNX format and the model in DL framework at inference time. During our research using the DL models listed, we did not find any discrepancies with the validity of the model in ONNX format.

[1]The CPU in this table is an AMD Threadripper 1950X. For more details see Section 3.5A.

When converting a model from PyTorch to ONNX, the conversion tools that are natively provided by PyTorch use a technique known as *tracing*. Tracing uses the dummy input, which the user is required to provide to convert a model, to capture the structure of PyTorch model by executing once and recording the flow of inputs through the model to construct the Torch IR graph to convert. This technique is stable and well supported. But it has some downsides including lack of broad support, especially since it only supports models that use limited control-flow (conditionals or loops). That means *tracing* does not capture models that use control-flow. It only captures sequences of operations that occur on a single execution as the dummy input flows through the graph on the PyTorch model. There is an alternative to *tracing* called *scripting* which supports all models because it converts syntax directly by first generating Python AST (Abstract Syntax Tree) and then using the JIT (Just In Time) compiler to do semantic analysis for conversion. This technique has several drawbacks including the need for code changes and it only supports a subset of python. The latter is a huge constraint because it means users might have to rewrite their python programs to comply with the requirements of conversion that uses *scripting*.

When using models with operators that may not be supported well, its important to use test data at inference time to make sure the outputs between DL framework and ONNX format are similar. Similar to what the converters does with the dummy input, which may not always be reliable, to ensure model consistency.

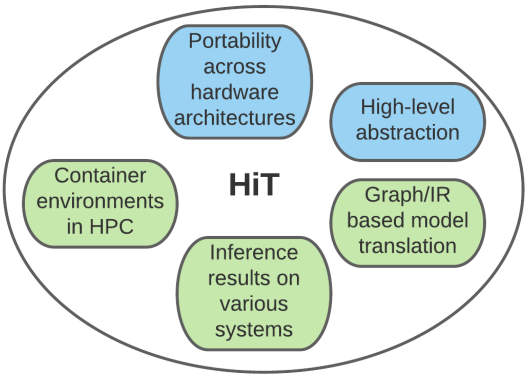

Figure 2: This figure shows the status of HiT. Green colored goals were demonstrated in this preliminary research and the blue colored goals will be completed in the next steps of this research.

5 Conclusion

This preliminary research started off with a goal of improving DL application workflows using high-level abstractions. Experimenting across many frameworks, systems, GPUs, and container softwares gave us the opportunity to learn about the advantages and disadvantages of various techniques that exist. All that experience helped us formulate a high-level abstraction that we are confident will improve the portability of scientific applications in HPC. We came across a number of potential techniques, some we incorporated into our abstraction, that will not only help DL applications in terms of portability but also in terms of performance.

6 Next steps

The immediate next goal for this research is to take the lessons we learned so far and apply them to increase CANDLE's portability. We will develop a working implementation of the high level abstraction, and explore the adaptability of containerization across more systems. Then the next goal would be to take the optimizations that ONNX Runtime applied for BERT which improved its performance and apply them to the CANDLE benchmarks to improve its performance across CPU and CUDA runtimes for NVIDIA GPUs and MiGraph for AMD GPUs. In addition, we also plan on run our experiments on the ARM based A64FX processor.

We also wanted to mention that most of the work done in this research uses ONNX Runtime focused on inference. We would, next, like to explore the ONNX Runtime for training. This is currently in beta so it would worth a study to explore what ONNX Runtime training offers.

References

[1] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. 2016. arXiv: 1603.04467 [cs.DC].

[2] David Brayford et al. "Deploying AI Frameworks on Secure HPC Systems with Containers." In: *2019 IEEE High Performance Extreme Computing Conference (HPEC)* (Sept. 2019). DOI: 10.1109/hpec.2019.8916576. URL: http://dx.doi.org/10.1109/HPEC.2019.8916576.

[3] Caviness. *Caviness*. 2021. URL: https://sites.udel.edu/it-rci/compute/community-cluster-program/caviness/.

[4] Tianqi Chen et al. *TVM: An Automated End-to-End Optimizing Compiler for Deep Learning*. 2018. arXiv: 1802.04799 [cs.LG].

[5] Ronan Collobert, Koray Kavukcuoglu, and Clement Farabet. "Torch7: A matlab-like environ- ment for machine learning". In: *BigLearn, NIPS Workshop*. 2011.

[6] *Crosscut Report: Exascale Requirements Reviews*. Tech. rep. The Office of Science, Mar. 10, 2017. URL: https://science.osti.gov/-/media/ascr/pdf/programdocuments/docs/2018/DOE-ExascaleReport-CrossCut.pdf.

[7] DARWIN. *Delaware Advanced Research Workforce and Innovation Network (DARWIN)*. 2021. URL: https://dsi.udel.edu/core/computational-resources/darwin/.

[8] Keras Developers. *Keras: API built on top of Tensorflow*. https://keras.io/. Accessed: 2020-05-07.

[9] NVIDIA Docker developers. *NVIDIA Container Toolkit*. NVIDIA. URL: https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/overview.html.

[10] ONNX Developers. *Open Neural Network EXchange*. https://onnx.ai. The Linux Foundation, 2019.

[11] ONNX Runtime Developers. *ONNX Runtime*. https://www.onnxruntime.ai. Microsoft.

[12] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL].

[13] *Early Application Results on Pre-exascale Architecture with Analysis of Performance Challenges and Projections*. Tech. rep. The Office of Science, Mar. 25, 2020. URL: https://www.exascaleproject.org/wp-content/uploads/2020/03/ECP_AD_Milestone-Early-Application-Results_v1.0_20200325_FINAL.pdf.

[14] *Frontier Center for Accelerated Application Readiness (CAAR)*. 2019. URL: https://www.olcf.ornl.gov/caar/frontier-caar/.

[15] Goswami, Manash and Palagiri, Kundana. *Deploying your Models to GPUs with ONNX Runtime in cloud and Azure Stack Edge*. [Online; accessed May 5th, 2021]. 2020. URL: http://developer.download.nvidia.com/video/gputechconf/gtc/2020/presentations/s21379-deploying-your-models-to-gpu-with-onnx-runtime-for-inferencing-in-cloud-and-edge-endpoints.pdf.

[16] Jeff Hale. "Deep Learning Framework Power Scores 2018". In: (2018). URL: https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a.

[17] Forrest N. Iandola et al. *SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and ¡0.5MB model size*. 2016. arXiv: 1602.07360 [cs.CV].

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. NIPS'12. Lake Tahoe, Nevada: Curran Associates Inc., 2012, pp. 1097–1105.

[19] K. Gregory Kurtzer, Vanessa Sochat, and W. Michael Bauer. "Singularity: Scientific container for mobility of compute". In: *PLOS One* (2017). DOI: 10.1371/journal.pone.0177459.

[20] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. "Numba: a LLVM-based Python JIT compiler". In: *LLVM '15: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. New York, NY: ACM, 2015, pp. 1–6.

[21] Dirk Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment". In: *Linux J.* 2014.239 (Mar. 2014). ISSN: 1075-3583.

[22] K. Jarrod Millman and Michael Aivazis. "Python for Scientists and Engineers". In: *Computing in Science Engineering* 13.2 (2011), pp. 9–12. DOI: 10.1109/MCSE.2011.36.

[23] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. arXiv: 1912.01703 [cs.LG].

[24] Fabian Pedregosa et al. *Scikit-learn: Machine Learning in Python*. 2018. arXiv: 1201.0490 [cs.LG].

[25] Philip C. Roth. *Developing Software for OLCF Frontier*. https://ecpannualmeeting.com/assets/overview/sessions/Roth-Path-to-Frontier-20200206.pdf. 2020.

[26] Reid Priedhorsky and Tim 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*. SC '17. Denver, Colorado: Association for Computing Machinery, 2017. ISBN: 9781450351140. DOI: 10.1145/3126908.3126925. URL: https://doi.org/10.1145/3126908.3126925.

[27] Vijay Janapa Reddi et al. *MLPerf Inference Benchmark*. https://arxiv.org/abs/1911.02549. 2020. arXiv: 1911.02549 [cs.LG].

[28] Frank Seide and Amit Agarwal. "CNTK: Microsoft's Open-Source Deep-Learning Toolkit". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, p. 2135. ISBN: 9781450342322. DOI: 10.1145/2939672.2945397. URL: https://doi.org/10.1145/2939672.2945397.

[29] *Summit Center for Accelerated Application Readiness (CAAR)*. 2018. URL: https://www.olcf.ornl.gov/caar/summit-caar/.

[30] Stéfan van der Walt, S Chris Colbert, and Gaël Varoquaux. "The NumPy Array: A Structure for Efficient Numerical Computation". In: *Computing in Science Engineering* 13.2 (Mar. 2011), pp. 22–30. ISSN: 1521-9615. DOI: 10.1109/mcse.2011.37. URL: http://dx.doi.org/10.1109/MCSE.2011.37.

[31] Justin M Wozniak et al. "CANDLE/Supervisor: A workflow framework for machine learning applied to cancer research". In: *BMC bioinformatics* 19.18 (2018), pp. 59–69.

[32] Justin M Wozniak et al. "Scaling deep learning for cancer with advanced workflow storage integration". In: *2018 IEEE/ACM Machine Learning in HPC Environments, MLHPC 2018*. Institute of Electrical and Electronics Engineers Inc. 2019, pp. 114–123.

[33] Yanzhao Wu et al. "A Comparative Measurement Study of Deep Learning as a Service Framework". In: *IEEE Transactions on Services Computing* (2019), pp. 1–1. DOI: 10.1109/TSC.2019.2928551.