1-13-2023

Modularizing Design Space Exploration And Optimizing Compilation-Time Of Deep Learning Workloads Using Tensor Compilers

Gaurav Verma
gaurav.verma@stonybrook.edu

Modularizing Design Space Exploration And Optimizing Compilation-Time Of Deep Learning Workloads Using Tensor Compilers

Research Proficiency Exam

Gaurav Verma

Supervisor: Dr. Barbara Chapman

Department of Computer Science

State University of New York at Stony Brook

January 2023

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this report are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this or any other university. This report is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text.

Gaurav Verma
January 2023

Abstract

Recently, server-edge-based hybrid computing has received considerable attention as a promising means to provide Deep Learning (DL) based services. However, due to the limited computation capability of the data processing units (such as CPUs, GPUs, and specialized accelerators) in edge devices, using the devices' limited resources efficiently is a challenge that affects deep learning-based analysis services. This has led to the development of several inference compilers, such as TensorRT, TensorFlow Lite, Glow, and TVM, which optimize DL inference models specifically for edge devices. These compilers operate on the standard DL models available for inferencing in various frameworks, e.g., PyTorch, TensorFlow, Caffe, and MxNet, and transform them into a corresponding lightweight model by analyzing the computation graphs and applying various optimizations at different stages. These high-level optimizations are applied using compiler passes before feeding the resultant computation graph for low-level and hardware-specific optimizations. With advancements in DNN architectures and backend hardware, the search space of compiler optimizations has grown manifold. Including passes without the knowledge of the computation graph leads to increased execution time with a slight influence on the intermediate representation. This report presents a detailed performance study of TensorFlow Lite (TFLite) and TensorFlow TensorRT (TF-TRT) using commonly employed DL models on varying degrees of hardware platforms. The work compares throughput, latency performance, and power consumption. The integrated TF-TRT performs better at the high-precision floating point on different DL architectures, especially with GPUs using tensor cores. However, it loses its edge for model compression to TFLite at low precision. TFLite, primarily designed for mobile applications, performs better with lightweight DL models than deep neural network-based models.

Further, we understood that benchmarking and auto-tuning the tensor program generation is challenging with emerging hardware and software stacks. Hence, we offer a modular and extensible framework to improve benchmarking and interoperability of compiler optimizations across diverse and continually emerging software, hardware, and data from servers to embedded devices. We propose HPCFair, a modular, extensible framework to enable AI models to be *Findable*, *Accessible*, *Interoperable* and *Reproducible* (FAIR). It enables users with a structured approach to search, load, save and reuse the models in their codes. We present our framework's conceptual design and implementation and highlight how it can seamlessly integrate into ML-driven applications for high-performance computing applications and scientific machine-learning workloads.

Lastly, we discuss the relevance of neural-architecture-aware pass selection and ordering in DL compilers. We provide a methodology to prune the search space of the phase selection problem. We use TVM as a compiler to demonstrate the experimental results on Nvidia A100 and GeForce RTX 2080 GPUs, establishing the relevance of neural architecture-aware selection of optimization passes for DNNs DL compilers. Experimental evaluation with seven models categorized into four architecturally different classes demonstrated performance gains for most neural networks. For ResNets, the average throughput increased by 24% and 32% for TensorFlow and PyTorch frameworks, respectively. Additionally, we observed an average 15% decrease in the compilation time for ResNets, 45% for MobileNet, and 54% for SSD-based models without impacting the throughput. BERT models showed an improvement with over 90% reduction in the compile time.

Table of Contents

Chapter 1

Introduction

The burgeoning applications of deep neural networks (DNN) are ubiquitous across multiple artificial intelligence domains, including industry and scientific disciplines. The deep neural architecture has evolved manifolds from simple neural networks to convoluted ones, followed by recurrent ones to massively large models such as Megatron-Turing Natural Language Generation (MT-NLG). Advancements in hardware such as GPU and TPU and DL frameworks like TensorFlow and PyTorch offer optimized kernel support facilitating DL innovations.

With an upsurge of deep learning computing processing units, deep learning-based object detection applications have received considerable research interest in recent years. In particular, as the accuracy of deep learning-based object recognition technology surpasses human capabilities, real-world deployment of intelligent surveillance technology with CCTV is expanding. Its use includes analyzing crowds and vehicle flows, performing fire detection and localization, and detecting unauthorized garbage dumping actions in urban regions. Edge computing (and inference at the edge) is also getting much attention in the scientific community, especially in High Energy Physics (HEP) [81]. For example, the next generation of HEP experiments, such as the High Luminosity Large Hadron Collider (HL-LHC) and Deep Underground Neutrino Experiment (DUNE), are investing in edge computing technology addressing real-time image analysis.

In most deployments, pre-trained deep learning models on servers are installed on edge devices located around sensors and provide services based on deep learning inference, such as object recognition while simultaneously performing data acquisition. The computing resources of edge devices are usually CPUs, GPUs, or FPGAs and have limited computing and power resources compared to cloud servers. Thus, various techniques and technologies have been developed to enable efficient deep learning inference on resource-constrained edge devices. These include the development of low-power, highly efficient SoC chips specialized for deep learning inference, such as Google's TPU and Intel's VPU, the development of model compression methods, such as quantization and the pruning of deep learning models for resource-constrained devices, as well as the design of lightweight models with reduced weights and parameters such as Mobile-Nets and YOLO, for use in edge computing environments. However, despite such advances, efforts are still needed to optimize deep learning applications like object detection models for computing as mentioned above environments significantly to maximize resource utilization. Several deep learning compilers, such as TVM, TensorFlow, TensorRT, and TensorFlow Lite, have been developed to address specialized accelerators' performance issues.

1.1 Overview of Tensor Compilers

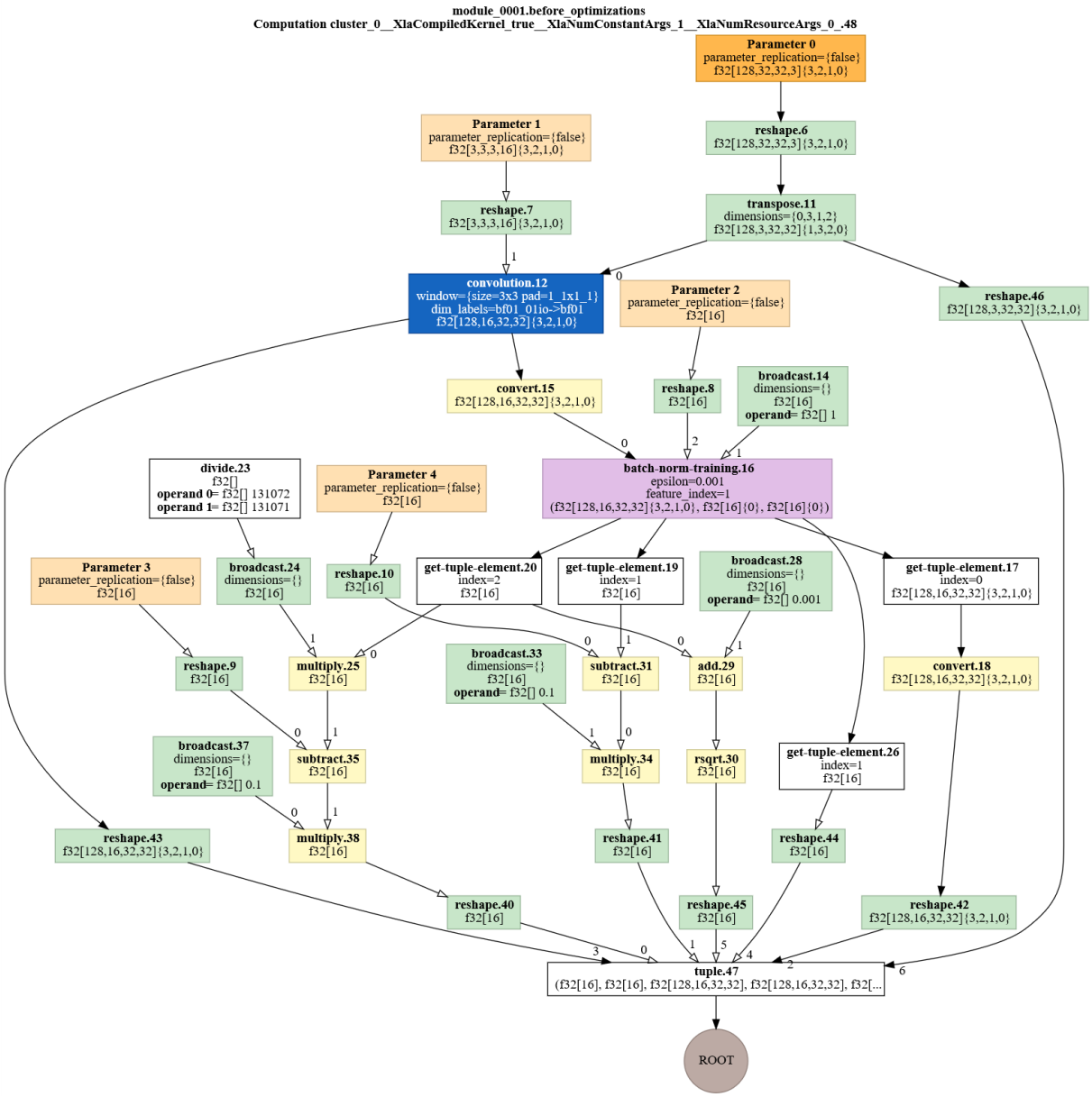


Figure 1.1: Computation Graph Before XLA Optimizations

The continuous effort from academia and industry has led to the development of many DL frameworks, including but not limited to TensorFlow [1], PyTorch [79], and MXNet [15]. These frameworks differ in design and implementation details. Hence, lack of interoperability hinders the reuse of the strengths offered by one by the other, leading to the duplication of engineering effort.

In the meantime, unique characteristics of the DL workloads have spurred the development of customized DL accelerators for higher efficiency, for example, Graphcore [48], SambaNova[28], Google TPU [49]. Apple Bionic, etc. We have general-purpose hardware (CPU, GPU), dedicated hardware (SambaNova Reconfigurable Dataflow Unit [28]), and emerging neuromorphic hardware (IBM TrueNorth [23]). With the emergence of hardware, it is critical to map the computation to the hardware efficiently. To perform matrix

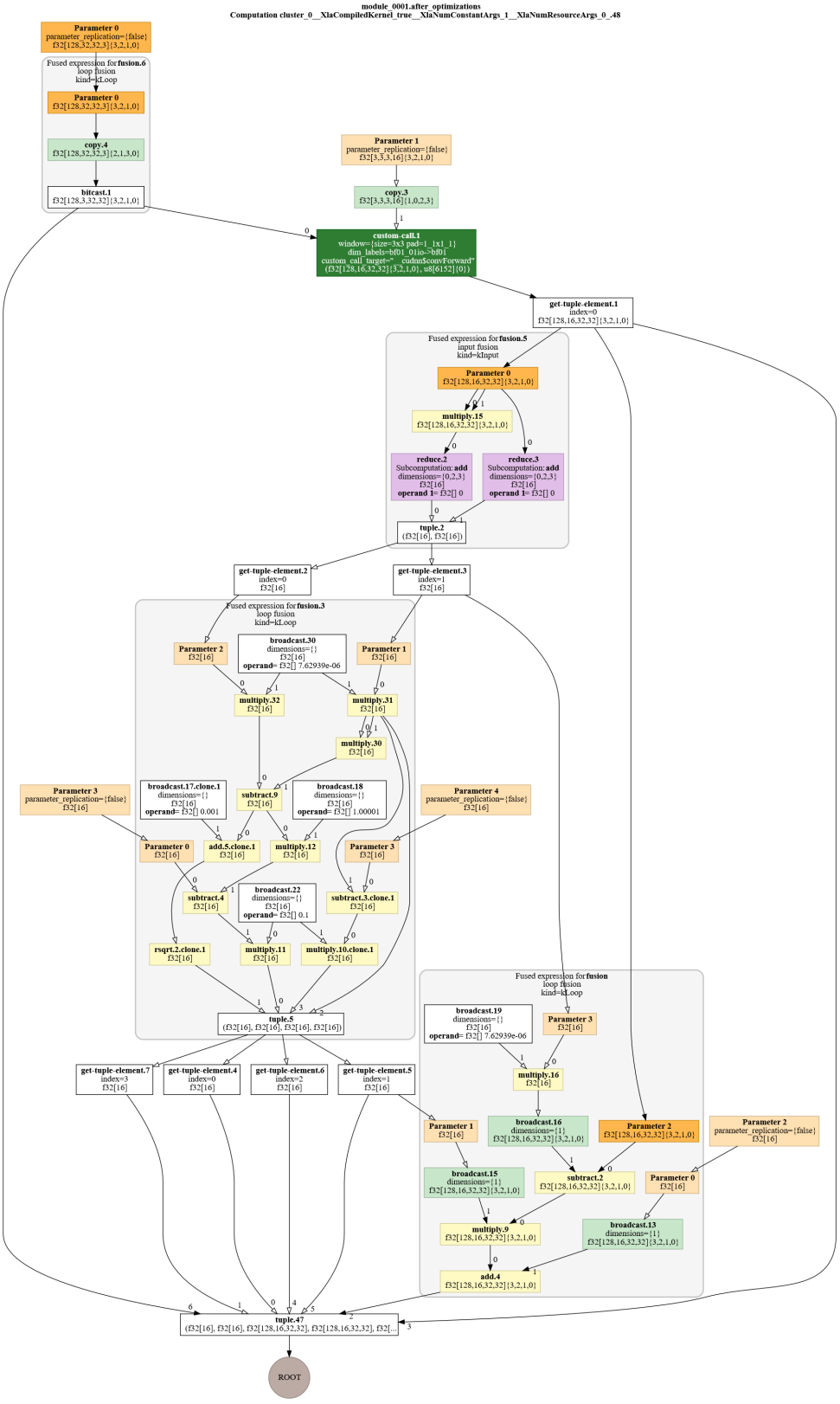


Figure 1.2: Computation Graph After XLA Optimization

multiplication, we have MKL and cuBLAS libraries. Similarly, hardware vendors offer hand-tuned libraries like MKL-DNN and cuDNN for the DL workloads. Again the challenges with the custom libraries are keeping up with the DL workload development and

the diversified hardware. The researchers have adopted domain-specific tensor compilers, like TVM, TensorRT, TFLite, XLA, and Glow, to address these limitations and challenges of manually optimizing for various hardware. The Tensor (or DL) compilers take model definition (computation graph) as an input, perform target-independent and dependent optimizations, and generate efficient code as an output to be deployed on the target device. Figure 1.1 and 1.2 present examples of computation graph optimizations taken from the high-level optimization graph of ResNet56 using Tensorflow XLA [96]. The gain in the throughput is over 2x after optimizations. The optimizations are applied as a sequence of transformations (schedules) incorporating the neural network and hardware information. Additionally, they employ a general-purpose compiler (LLVM) for portability across various hardware platforms. Analogous to traditional compilers, tensor compilers adopt the layered architecture, including frontend, intermediate representations (IR), and backend. They differ in the design of multi-level IRs and domain-specific optimizations.

1.2 Chapter Organization

The rest of the report is organized as follows: Chapter 2 discusses the optimizations offered by TensorRT and TFLite and compares their performance on the architecturally varying hardware. It evaluates the compilers on the metrics like throughput, latency, reduction in the model size, and power consumption. Further, chapter 3 presents our framework to modularize the DL components, helping in efficient search space exploration of the optimization space. Lastly, chapter 4 examines the neural architecture-aware optimizations to reduce the compilation time. We conclude this report in chapter 5, summarizing the work accomplished so far and discussing future work.

Chapter 2

Performance Evaluation of Tensor Compilers For Inference

2.1 Primary Contributions

The main contributions of this work [105] are summarized as follows:

- The work presents a detailed performance analysis of TensorFlow Lite and TensorFlow TensorRT inference compilers by comparing throughput, latency, and power consumption.
- The work describes inference compilers' performance behavior concerning DL model architecture and hardware.
- The work reveals a need for a standardized benchmark suite to analyze the performance of inference compilers' optimization pipeline for edge computing.

2.2 Motivation

Deep Neural Network (DNN) based Machine Learning applications are gaining popularity to enable Artificial Intelligence on edge. For instance, in an object detection scenario, a DNN extracts feature from the input images and classify them from the predefined categories. The inference process in a trained DNN model relies on a forward pass. It is computation-intensive, challenging the constrained computing resources typically available on edge devices. In a cloud data center-based approach to handling this, the input data gathered through edge devices are offloaded to the cloud for processing. The results are returned to the edge devices after the inference has been performed. High latency, high memory requirements, poor real-time performance in throughput and power consumption, and poor user experience typically limit these implementations.

Methods [38] have been developed to distribute the DNN's computations between the cloud and the edge. Although these approaches, e.g., an early exit [52] , provide a reduction in latency, they have implementation challenges for certain types of DNN. For example, AlexNet, with over 60 million nodes, is hard to partition in real-time.

The inference-on-edge paradigm has emerged as a solution to address the problems of cloud-based inferencing. It aims to bring computing close to the data source to reduce

latency, bandwidth use, and power consumption. Several techniques have been proposed to enable inference on edge, including model redesign [115] and human-invented architecture [110] along with model compression solutions [11], network pruning [59], parameter quantization [24], hardware acceleration based on parallel computing [42], and software acceleration focused on optimizing resource management and pipeline design. System on a Chip (SoC) [108] designs are another notable effort to improve the efficiency of inference on edge. However, the lack of any standardized chipset [111] inhibits any general optimization.

Extreme diversity in hardware and software technologies makes the development of a framework that can optimize DNN inference models for all edge systems challenging, especially with regard to low-level optimizations. The TF-TRT integrated solution and TFLite are designed with the intention of being hardware agnostic. These frameworks take as input DNN models from DL frameworks like Tensorflow, PyTorch, Caffe2, ONNX, etc., and perform fine-grained optimizations viz., quantization, layer, and tensor fusion, along with computation graph-based optimizations, delivering a lightweight model to be deployed on the edge devices. They perform these optimizations with the goal of improving memory management, power consumption and GPU utilization. Section 2.3.1 discusses these frameworks in detail.

2.3 Evaluation

2.3.1 Compilers Under Test (CUT)

TensorFlow-TensorRT Integrated Solution

TensorRT (TRT) is a CUDA-based SDK for high-performance deep learning inference. It optimizes the inference and provides a runtime that delivers low latency and high-throughput for deep learning inference applications. The tight integration of TRT with TensorFlow (TF) makes the use of high-performance inference engines possible. It facilitates TRT optimizations to the TF models by optimizing the supported graphs, leaving unsupported operations to be executed by TF. TRT scans the TF graph to identify the sub-graphs, selecting candidates for graph partitioning based on the supported operations. After identifying such candidates, it converts TF layers into TRT layers for each sub-graph. Subsequently, these sub-graphs are converted to optimized TRT engines, as explained below, replacing the existing TF sub-graph. TRT's optimizations can be ported to other Nvidia GPUs. The TRT-specific optimizations are only supported on Nvidia GPUs, restricting it from being leveraged on different back-end hardware iOS GPUs or Adreno.

The ability to take diverse DL models as input makes TRT applicable to a wide range of AI based edge applications. For example, TRT can execute a variety of computer vision models to guide an unmanned aerial system flying in dynamic environments autonomously. It can be embedded to enable high throughput execution of neural network models in autonomous vehicles, deliver video analytics at the edge, or the data center, such as NVIDIA's DeepStream [75]. TRT also provides an ONNX parser and runtime extension to frameworks like Caffe 2, MxNet, Chainer, Microsoft Cognitive Toolkit, and PyTorch.

Pivotal Optimizations in TensorRT

TensorRT offers `INT8` and `FP16` reduced precision calibration. Following training in 32-

bit precision (F32), the inference can employ half-precision, `FP16`, or even `INT8` tensor operations because gradient back-propagation is not done during the inference phase. Lower precision helps in achieving a smaller model size, lower memory utilization and latency, and higher throughput. We can control the precision in TensorRT by specifying the Data Type in the `uff_to_trt_engine` function. Since `INT8` precision can represent only `256` values (-128 to 127), TensorRT performs calibration to represent the weights and activation as 8-bit integers minimizing the accuracy loss. The calibration is a fully automated and non-parameterized method to convert `FP32` to `INT8` using a representative input training data sample.

Additionally, it performs layer and tensor fusion by parsing the computation graph and performing graph optimizations. The graph optimizations make the execution efficient without changing the underlying computation. Ordinarily, the kernel computation is faster than the kernel launch overhead coupled with the cost of reading and writing the tensor data for each layer. TensorRT addresses the memory bandwidth bottleneck and under-utilization of available GPU resources by vertically fusing the kernels to perform the sequential operations within a single kernel launch. TensorRT identifies the layers with common input data and filter size with different weights. Further, it performs horizontal fusion to convert them into a single kernel to avoid launching more than one kernels. In short, this results in an efficient graph with fewer layers and kernel launches, reducing inference latency.

TensorRT also supports kernel auto-tuning by selecting the most suitable data layers and algorithms for each target GPU platform. For example, based on the target GPU, input data size, filter size, tensor layout, batch size, and other important parameters, the inference engine opts for the best convolution algorithm from the kernels library to perform the task. Furthermore, TRT allows dynamic tensor memory management, which allocates memory for a tensor during its life-span only, thereby reducing memory footprint and enhancing memory reuse.

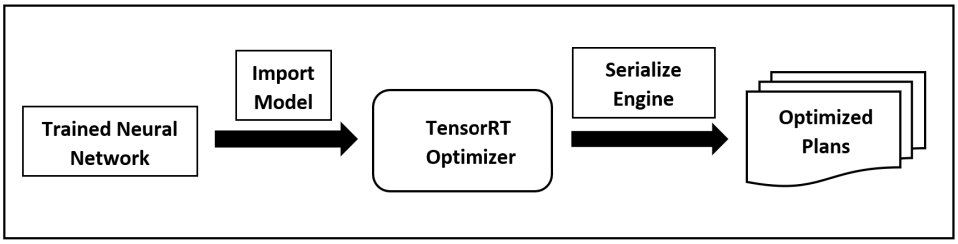


Figure 2.1: Import and optimize trained models to generate inference engines

Consider Figure 2.1. The models trained on well-known frameworks like TensorFlow, Caffe2, MxNet, and PyTorch are fed to TRT. TRT then produces a light-weight runtime engine after optimizing the neural network computation for parameters like batch size, precision, and workspace memory for the target deployment GPU. The generated engine is an optimized inference execution engine serialized to a plan file.

The TF-TRT workflow involves three phases as shown in Figure 2.2. In the first phase, a model is designed, developed, and trained in the supported frameworks. A deployment solution is then detailed and validated as part of the second phase. Factors taken into account while devising the deployment solution include: whether it is based on a single network (object detection) or multi-network (federated learning), computing device, data ingestion pipeline, data format and nature, and so on. Figure 2.2 shows that once the

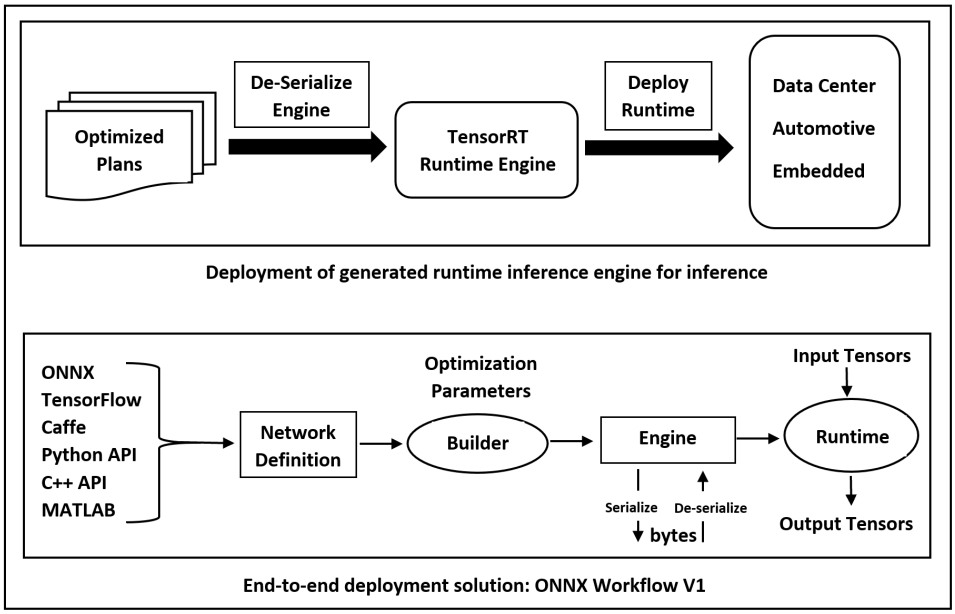


Figure 2.2: Deployment Workflow

inference architecture is decided, the next step is to build the inference engine using TRT. The trained model is fed to the deployment pipeline using ONNX parser, Caffe Parser, or UFF Parser. Like ONNX, a UFF package offers utilities to convert and parse trained models from differing frameworks to a standard format. Following this, the TRT builder applies various optimization parameters, to say, batch size, mixed precision, and many more, to build an optimized inference engine specific to the infrastructure. The output inference engine is validated for accuracy as INT8 and FP16 based quantizations might lead to a slight precision loss. It is subsequently written to a plan file in a serialized format. The inference engine is initialized by deserializing this model from the plan file into an inference engine. In the last phase, the TensorRT library is linked to the deployment pipeline and is asynchronously called on-demand.

TensorFlow Lite

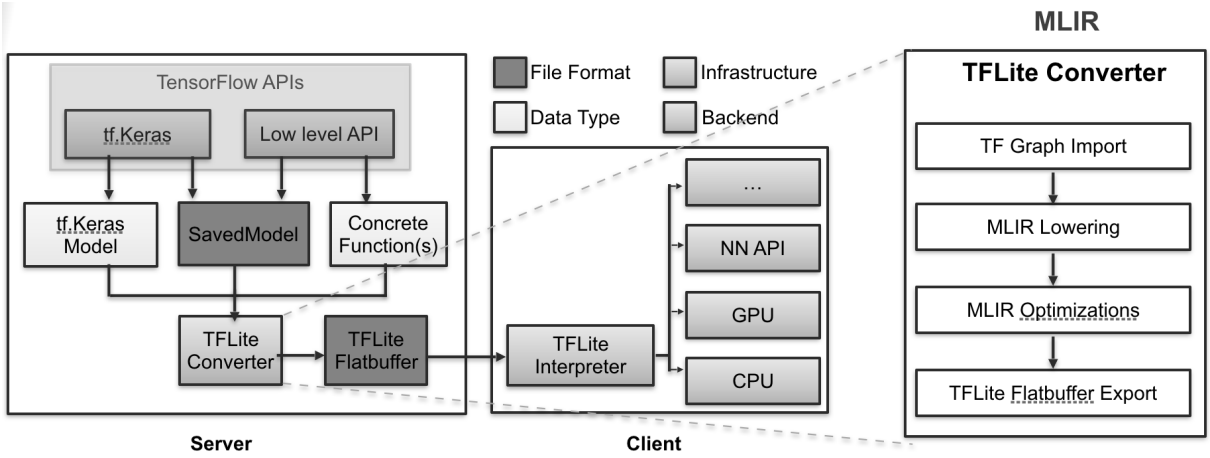


Figure 2.3: TensorFlow Lite Architecture

Google's TensorFlow Lite [34] provides a set of developer tools to leverage DL models' potential on edge devices. As shown in Figure 2.3, TFLite consists of two main components: the server-side MLIR-based TFLite Converter and the client-side TFLite Interpreter. Multi-Level Intermediate Representation (MLIR) [51] is the reusable and extensible compiler framework for defining compiler IRs, from high-level (such as DL framework specific) to low-level LLVM-IR.

The TFLite converter tool takes a trained model in a standard format such as TensorFlow or a format interconvertible by ONNX and generates a TFLite model file (.tflite). After conversion, the model file can be deployed to a client device (e.g., a mobile or embedded system) and run locally using the TFLite interpreter. TFLite supports inference on mobile and embedded platforms, such as Android, iOS, and Linux (including Raspberry Pi), in multiple programming languages.

Edge devices are severely resource constrained, hence TFLite performs specific optimizations to generate lightweight models targeting size and latency reduction. It supports various optimization techniques, such as quantization, pruning, and clustering. By default, the TFLite interpreter utilizes CPU Kernels explicitly optimized for the ARM Neon Instruction set [4]. To fully use the back-end hardware, including accelerators like GPU, TFLite offers delegates' support.

A delegate acts as a bridge between TFLite runtime and lower-level APIs associated with accelerators like OpenGL/OpenCL for Mobile GPUs. Among numerous delegates offered by TFLite, GPU Delegate is optimized for Android and iOS. It is essentially optimized to perform floating-point matrix operations, allowing an interpreter to execute GPU-supported operations on the device's GPU.

At the time of this study, GPU delegate supports 23 TF operations, e.g., `ADD, EXP` [34]. These operations help to optimize the performance on accelerators compared to the CPUs' execution solely. It is crucial to confirm the model's supported operations before choosing a delegate. Many unsupported operations can lead to multiple hops between CPU and accelerator, impacting the latency adversely. Further, using a delegate entails added trade-offs, e.g., using a GPU Delegate induces overhead during initialization. Also, the GPU delegate does not support the quantized model.

Pivotal Optimizations in TensorFlow Lite

During the model conversion from a trained model to TFLite flat buffer format, TFLite Converter supports four types of quantization; dynamic range, integer, float16, and mixed precision. Dynamic range quantization supports on-the-fly quantization and dequantization of activations so that quantized kernels can be utilized when available. Integer quantization is an optimization strategy that converts 32-bit floating-point numbers (such as weights and activation outputs) to the nearest 8-bit fixed-point numbers. Float16 quantization results in a 2x reduction in model size in exchange for minimal impacts to latency and accuracy. Mixed precision converts activations to 16-bit integer values and weights to 8-bit integer values. This mode can significantly improve the quantized model performance when activation is sensitive to the quantization while still achieving an almost 3-4x reduction in model size.

Moreover, the fully quantized model can be consumed by integer-only hardware accelerators. TFLite has exposed APIs to configure and prune either the entire model or a few selected layers to support pruning. The pruning works by removing parameters within a model that have only a minor impact on its predictions. This technique brings improvements via model compression. Another optimization that the TFLite converter performs

is fusion of operations (various tensor computations). Fused operations maximize the performance of their underlying kernel implementations and provide a higher-level interface to define complex transformations like quantization. TFLite also supports clustering as an optimization. The clustering works by grouping each layer's weights in a model into a predefined number of clusters, then sharing the centered values for the weights belonging to each cluster. Hence it reduces the number of unique weight values in a model and reduces its complexity.

2.3.2 Dataset

This section summarizes the statistical information corresponding to the datasets used by the pre-trained models to evaluate the frameworks in this work.

ImageNet

The ImageNet dataset is a collection of human-annotated images organized according to the WordNet hierarchy and designed for developing computer vision applications such as image classification, object detection, and object localization. WordNet is a database of English words associated with semantic relationships. Each meaningful concept in WordNet, perhaps described by multiple words or word phrases, is called a "synonym set" or "synset." ImageNet offers variations of the same object, including camera angles and lighting conditions. As per the ImageNet homepage [43], more than 14 million images are organized into over 21,000 subcategories averaging around 500 images per subcategory. These categories are subcategories of 21 high-level categories. Additionally, slightly over 1 million images have been annotated with the bounding boxes. There are 1000 synsets and 1.2 million images with Scale-Invariant Feature Transform (SIFT) features. SIFT helps in detecting local features in an image.

Common Objects in Context (COCO)

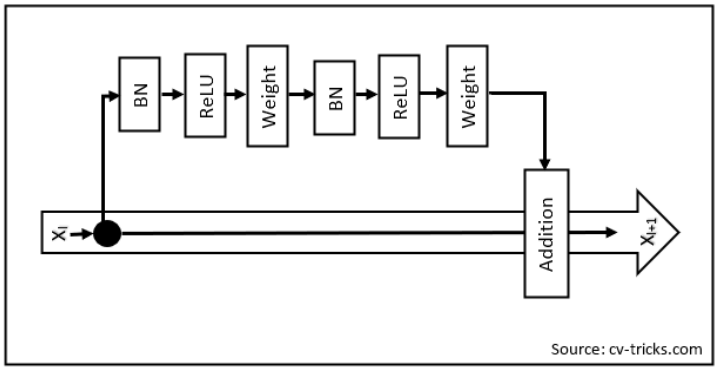
Microsoft's Common Objects in Context [56] is a large-scale object detection, segmentation, and captioning dataset. It consists of everyday scenes comprising common objects in their natural context. Objects are labeled using per-instance segmentations to aid in precise object localization. There are 165,482 train, 81,208 validation, and 81,434 test images encompassing 91 categories. The major portion of the dataset is non-iconic images, as they are better at generalizing.

2.3.3 Evaluated Models

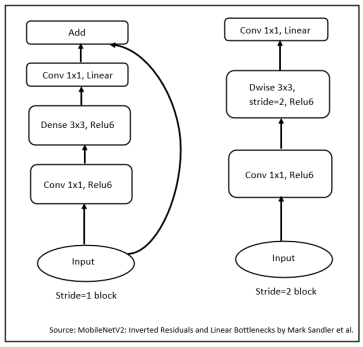
This section discusses the DL models used in the work for performance evaluation. We focused on Image Classification and Object Detection tasks. For the Image Classification, we used pre-trained Keras models, viz., ResNet50_v2, and MobileNet_v2, trained on the ImageNet dataset. We used the TF implementation of the SSD-MobileNet_v2 model available from the MLPerf Benchmark [27], trained on COCO Dataset to perform Object Detection.

The ResNet50 is a 50-layer deep convolutional neural network (CNN). The Residual Networks (ResNet) are similar to any other deep network with convolution, pooling, activation, and fully-connected layers, except for the identity connection between the layers. The identity connection links the input to the end of the residual block. The

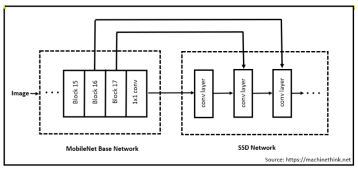
residual block gets its name because, unlike other networks, it tries to learn the residue instead of the output. The ResNet50 inputs an image and performs convolution and max-pooling. Then it passes through four phases. In the first phase, three residual blocks contain three layers each, performing the convolution with stride 2. The stride of 2 is responsible for making the input size half and doubling the channel's height and width. As it progresses to later stages, the channel width will double, halving the input size. As shown in Figure 2.4a, before the multiplication with the weight matrix, it applies Batch Normalization and ReLU activation to the input tensor. Batch Normalization increases the network's performance by adjusting the input layer.



(a) Basic architecture of ResNet50_v2



(b) Basic architecture of MobileNet_v2



(c) Basic architecture of SSD_MobileNet_v2

Figure 2.4: Basic architecture of the evaluated models

We also used MobilNet_v2. It is a lightweight CNN-based model introduced by Google, quite suitable for edge devices due to its size. It is derived from an inverted residual structure where the residual connections are between the bottleneck layers. The intermediate expansion layer uses lightweight depth-wise convolutions to filter features as a source of non-linearity. As shown in Figure 2.4b, generally, the first layer is a 1x1 convolution with ReLU6. The second layer is the depth-wise convolution. And lastly, the third layer is another 1×1 convolution but without any non-linearity. It contains an initially fully connected layer with 32 filters and 19 residual bottleneck layers.

We also used the SSD_MobileNet_v2 model for the work. The SSD stands for Single Shot MultiBox Detection. The Single Shot refers to the object localization and classification tasks performed in a single forward pass of the network. Multibox is the name of the technique for bounding box regression developed by Wei Liu et al. [58]. The model's input is a single image of 1x3x300x300 (BHWC) in RGB order. It is further converted to BGR format internally. The output is a typical vector containing the tracked object data. As shown in Figure 2.4c, the initial network consists of standard MobileNet architecture

truncated before any classification layers, termed as the base network. An auxiliary network follows it; called the SSD network. The SSD network is based on a feed-forward convolutional network consisting of feature maps extraction and Object Detection using a convolution filter. It produces a fixed-size collection of bounding boxes and scores (probabilities) for the respective object classes present within those boxes.

2.3.4 Hardware Specifications

We carried out our experiments on two different Turing architecture-powered NVIDIA GPUs, GeForce RTX 2080 and Tesla T4. In our experimentation setup, where Tesla T4 GPU uses tensor cores, RTX 2080 GPU does not. The tensor cores offered by Tesla T4 help accelerate certain half-precision matrix algebra types, including General Matrix Multiplication (GEMM). It also enables faster and easier mixed-precision computation. We can control the automatic mixed-precision by setting/unsetting the environment variable, `TF_ENABLE_AUTO_MIXED_PRECISION`. RTX 2080 has a core clock speed of 1515 MHz and a Power Consumption (TDP) of 215 Watt. On the other hand, Tesla T4 has a core clock speed of 585 MHz and power consumption (TDP) of 70 Watt. The clock speed of RTX 2080 is around 35% higher than Tesla T4, whereas typical power consumption in Tesla T4 is 3x times lower than RTX 2080. Both have PCIe 3.0 x16 interface and have GDDR6 memory types. The maximum RAM amount for RTX 2080 is 8 GB, whereas Tesla T4 offers 16 GB. Higher RAM enables Tesla T4 to store more working data and machine code currently in use, allowing quick-access and faster data processing. Additionally, the peak memory clock speed in RTX 2080 is 14000 MHz compared to 10000 MHz in Tesla T4. The better memory clock speed offers faster data read and store in the case of RTX 2080. We experimented with the `per_process_gpu_memory_fraction` flag set to 0.7. That means TensorFlow (TF) allocates a maximum of 70% of GPU memory for all its internal usage.

For the TF-Lite, we experimented with `num_threads` set to 1, 4, and 8 in the interpreter. TF-Lite by default uses the maximum number of threads available which drastically impacts the performance due to GPU resource constraints. During our experimentation, we also found that with the number of threads set to more than 8, the power consumption increased. We used the optimal number of threads based on our experiments and set `num_threads` to 4 for all our TF-Lite experiments.

2.3.5 Software Specifications

We carried out the TensorFlow Lite (TF-Lite), and TensorFlow-TensorRT (TF-TRT) experiments with the following software stacks:

For the experiments with TF-Lite, we have used TF-Lite v2.3, TensorFlow 2.4, CUDA 10.1, and CuDNN v7.5. For the experiments with TF-TRT, TensorRT's version was v5.1.5.0, TensorFlow-GPU v2.0, CUDA 10.1, and CuDNN v7.5. Since TFLite is not entirely optimized for `x86` architecture and desktop GPUs, we also carried out the experiments on a simulated device using Android Studio Emulator. We used Android Studio 4.0.1 and Pixel 3a XL simulated devices with android 10, and API 29.

2.3.6 Evaluation Metrics

We selected the following metrics to compare the performance of edge inference using TF-TensotRT and TFLite:

- **Throughput:** the volume of inferences within a given period, usually measured in inferences per second or samples per second (imgs/sec).
- **Latency:** the execution time to perform inference on one image, expressed in milliseconds (ms).
- **Power:** refers to the power drawn by the GPU to perform one inference. It is expressed in Watt (W),
- **Model Size:** the saved model's (.pb or .tflite) size on the disk. It is measured in Megabyte (MB).

2.4 Discussion

2.4.1 Experimentation on GeForce RTX 2080

Table 2.1 presents the results from the experiments conducted on GeForce RTX 2080 GPU. For the precision mode `FP32` and `FP16`, the TF-TRT optimized ResNet50 model showed a 2x increase in the throughput compared to the native model. In the case of `INT8` and `MIXED` precision mode, the latency has improved significantly by 8x, boosting the throughput by nearly 7x-8x. This behavior is attributed to the TF-TRT's horizontal and vertical fusions that reduce the number of kernel launches. Across all the precision modes, TF-TRT could reduce the power consumption in a range of 10%-30%. It is noted that the saved model's size has significantly increased after the optimizations by TF-TRT, leaving the size of parameters (assets and variables) the same. We found that the induced optimized engines saved copies of the weights and variables during experiments to be later used to inspect the saved model for an efficient execution plan selection. The above finding supports the 8x times increase in the model's size post optimizations by TF-TRT. We could see a similar behavior of TF-TRT with the MobileNet too.

On the contrary, TFLite could not perform well with a deep CNN-based model like ResNet50 on `x86-64` architecture. TFLite is not optimized for desktop-based GPUs and could not utilize GPU delegate to its potential. Hence, most of the computations are executed on the CPUs, leading to an over 95% drop in the throughput. Irrespective of that, TFLite showed a significant decrease in the model size by 75% and an overall reduction in power consumption of roughly 35%. With the decrease in the precision mode, the space needed to save the model also reduced proportionally.

Discussing the results obtained from the experiment with the MobileNet, TF-TRT showed behavior similar to the ResNet50 model. Since MobileNet is not as deep as ResNet50, TFLite performs comparatively better against its performance with the ResNet50 model. Still, we can see that almost all the computation is getting executed on the CPU as it cannot efficiently employ GPU delegate on a `x86-64` architecture. Notwithstanding that we experimented with multi-threaded settings and optimized data format, NCHW, the results exhibited it did not improve the performance much.

However, SSD_MobileNet is a different architecture. The results reveal a decrease in the model size with TF-TRT. There is an 8x-11x increase in the throughput and a 15%-35% reduction in the power consumption in TF-TRT. Talking about the TFLite, despite a reduction in power consumption and the model size, overall, TFLite does not perform on par with TF-TRT. TFLite displays this behavior because it is not optimized for the GPUs

Framework	Precision	Avg_Throughput(imgs/sec)	Avg_Latency (ms)	Avg_Power(W)	Model_Size(MB)
Native	FP32	244.50	4.089979	48	98
TF-TRT	FP32	405.73	2.464693	42	200
TFLite		5.07	197.238658	34	98
TF-TRT	FP16	485.68	2.058968	37	200
TFLite		5.10	196.078431	35	49
TF-TRT	INT8	1469.84	0.680346	34	200
TFLite		2.55	392.156862	34	25
TF-TRT	MIXED	1777.78	0.562499	34	200
TFLite		2.51	398.406374	34	25

(a) Resnet50_v2 model trained on ImageNet Dataset.

Framework	Precision	Avg_Throughput(imgs/sec)	Avg_Latency (ms)	Avg_Power(W)	Model_Size(MB)
Native	FP32	345.29	2.896116	65	14
TF-TRT	FP32	778.60	1.284356	57	32
TFLite		32.71	30.571690	35	14
TF-TRT	FP16	1326.91	0.753630	61	32
TFLite		32.91	30.385900	33	7
TRT	INT8	1803.46	0.554489	43	32
TFLite		18.53	53.966540	34	4
TF-TRT	MIXED	2320.10	0.431015	42	32
TFLite		18.68	53.533190	35	4

(b) MobileNet_v2 model trained on ImageNet Dataset.

Framework	Precision	Avg_Throughput(imgs/sec)	Avg_Latency (ms)	Avg_Power(W)	Model_Size(MB)
Native	FP32	50.72	19.715548	63	67
TF-TRT	FP32	394.31	2.536058	54	65
TFLite		8.89	112.485939	42	14
TF-TRT	FP16	399.15	2.505302	53	65
TFLite		9.28	107.758620	42	7
TF-TRT	INT8	476.57	2.098322	49	45
TFLite		4.90	204.21268	43	4
TRT	MIXED	624.43	1.601458	42	45
TFLite		4.92	204.081632	42	4

(c) SSD_MobileNet_v2 model trained on COCO Dataset.

Table 2.1: Comparison between TF-Lite and TF-TensorRT on GeForce RTX 2080 GPU

Framework	Precision	Avg_Throughput(imgs/sec)	Avg_Latency (ms)	Avg_Power(W)	Model_Size(MB)
Native	FP32	180.76	5.532197	64	98
TF-TRT	FP32	458.62	2.180454	38	200
TFLite		6.97	143.472022	43	98
TF-TRT	FP16	1373.41	0.728114	37	200
TFLite		4.81	207.900207	39	49
TF-TRT	INT8	2207.10	0.453083	40	200
TFLite		2.47	404.858299	33	25
TF-TRT	MIXED	2353.43	0.424911	39	200
TFLite		2.46	406.504065	32	25

(a) Resnet50_v2 model trained on ImageNet Dataset.

Framework	Precision	Avg_Throughput(imgs/sec)	Avg_Latency (ms)	Avg_Power(W)	Model_Size(MB)
Native	FP32	409.00	2.444987	43	14
TF-TRT	FP32	1584.27	0.631205	33	32
TFLite		39.56	25.278058	23	14
TF-TRT	FP16	2598.20	0.384881	34	32
TFLite		39.98	25.012506	24	7
TF-TRT	INT8	3883.14	0.257523	31	32
TFLite		18.79	53.219797	19	4
TF-TRT	MIXED	3509.76	0.284919	30	32
TFLite		18.83	53.106744	20	4

(b) MobileNet_v2 model trained on ImageNet Dataset.

Framework	Precision	Avg_Throughput(imgs/sec)	Avg_Latency (ms)	Avg_Power(W)	Model_Size(MB)
Native	FP32	32.12	31.133251	43	67
TF-TRT	FP32	402.49	2.493765	38	65
TFLite		8.57	116.686114	29	65
TF-TRT	FP16	423.93	2.375296	37	65
TFLite		8.57	116.67645	28	33
TF-TRT	INT8	610.29	1.642036	39	45
TFLite		4.75	210.526315	26	17
TF-TRT	MIXED	654.12	1.524390	37	45
TFLite		4.76	210.084033	25	17

(c) SSD_MobileNet_v2 model trained on COCO Dataset.

Table 2.2: Comparison between TF-Lite and TF-TensorRT on Tesla T4 GPU

installed on the `x86-64` architecture. TFLite interpreter utilizes CPU Kernels explicitly optimized for the ARM Neon instruction set, not for `x86-64` instruction set.

2.4.2 Experimentation on Tesla T4

We repeated the experiments discussed in the last section on a tensor core GPU, Tesla T4. The results are summarized in Table 2.2. As expected, overall, TF-TRT and TFLite demonstrated similar behavior for all three models. TF-TRT successfully optimized the inference latency, throughput, and power consumption. However, it suffers from an expansion in the model's size for the reasons above. Being that TFLite is optimized for ARM-based GPUs. We see a drastic reduction in the throughput for all three TFLite-optimized models. Still, it can reduce the model size significantly by a factor of 75% for `INT8` precision mode.

We also observed that the throughput improved by 1.3x-1.6x on a tensor core GPU for the TF-TRT optimized models compared to a non-tensor core GPU. It was the case, especially for `FP16`, `INT8`, and `MIXED` precision mode. The tensor cores are activated when the parameters of the layers are divisible by 8 or 16. Also, on a Tesla T4 machine, the GPU utilization increased extensively. Compared to the native model's utilization of a maximum of 60% of the available GPU resources, the TF-TRT optimized model utilized over 95% of the available resources on a tensor core GPU. TF-TRT's selection of hand-tuned libraries and lowering kernel launches lead to lowered GPU wait or idle time.

Backend	Model	Precision	Avg_Throughput(imgs/sec)	Avg_Latency (ms)	Model_Size(MB)
GPU	MobileNet_v1	Floating	745.57	10.73	17
CPU*			311.65	25.67	
GPU	MobileNet_v1	Quantized	NS	NS	NS
CPU*			571.43	14.35	4.1
GPU	SSD_MobileNet	Floating	41.67	24	27
CPU*			18.86	53	

*4 Threads; NS: Not supported

Table 2.3: Execution of TF-Lite models on an android device

2.4.3 Experimentation on Pixel 3a XL, an android device

Due to the lack of any standard benchmark that can evaluate both frameworks to provide an apple-to-apple comparison, we conducted experiments on an Android Device to assess the performance of TFLite. Table 2.3 shows our experimental results on Pixel 3a XL, a simulated device with android 10. The GPU delegate allows TFLite to utilize the available GPU resources. The GPU delegate executes the GPU-compatible operators, and the CPU executes the remaining operations. There was a 2x increase in the throughput for both the TFLite-optimized floating models compared to the native models.

Further, the GPU delegate does not support quantized models on android yet. It causes computations to be executed on the CPU. But still, TFLite performs comparatively

with a 75% reduction in the model's size.

We have also estimated the loss in precision for all of the experiments conducted due to the various optimization techniques specific to individual frameworks. Conclusively, TF-TRT-based optimizations lead to less than 10^{-5} precision loss for image classification tasks for the precision mode `FP32` and `FP16`. For `INT8` and `MIXED` there was a loss to an extent of 10^{-3}. Similarly, for TFLite, there was a loss in precision due to quantization by a factor of 10^{-3} for `F16` and lower precision mode. We computed the Mean Average Precision for the native and optimized model for the Object Detection task. The results for TF-TRT, TFLite, and native TF-based models were on par. To summarize, we can say that the loss in precision due to the stated compilers' various optimizations is in the acceptable range.

There is a clear need to optimize DL models on edge devices for better latency and power performance. TensorFlow Lite (TFLite) and TensorFlow-TenorRT (TF-TRT) are considered state-of-the-art inference compilers for edge computing. This chapter presents a detailed performance study of TFLite and TF-TRT using commonly employed DL models for edge devices on varying hardware platforms. We find that TF-TRT integration performs better at high precision mode but loses its edge for model compression to TFLite at low precision mode. TF-TRT consistently performs better with different DL architectures, especially with GPUs using tensor cores. However, TFLite performs better with lightweight DL models than with deep neural network-based models.

The scientific computing community is considering edge computing for its ML workload to analyze real-time data during experiments. The performance of inference compilers such as TensorRT and TensorFlow Lite under a scientific ML workload is an essential question for the scientific community. As a next step, we plan to evaluate the performance of these frameworks using the DOE FAIR (Findable, Accessible, Interoperable, and Reusable) workload [25] for modern AI accelerators such as Vision Processing Units (VPU), Field-Programmable Gate Array (FPGA), Application-Specific Integrated Circuits (ASIC), and Tensor Processing Unit (TPU).

2.5 Related Work

Compiler development for DNNs has been spotlighted in the modern era of Machine Learning. Apache TVM [14], Facebook's Glow [29], Intel's nGraph [21], Nvidia's TensorRT [76], Google's XLA [96] and Tensorflow Lite [34] are a few notable frameworks designed to compile deep learning models into minimum deployable modules. These frameworks accept a computation graph from deep learning frameworks, such as PyTorch, Caffe2, and Tensorflow, and generate highly optimized code for machine learning accelerators.

The comprehensive survey by Mingzhen Li et al. [53] presents the DL compilers' unique design architecture. It emphasizes the DL-oriented multi-level IRs, and front-end/back-end optimizations in TVM, nGraph, TC, Glow, and XLA. Nevertheless, it does not discuss TensorFlow-TensorRT, Tensorflow Lite, or compilers for edge inference. TF-TRT and TFLite provide a framework for applying fine-grained optimizations to any input DNN models employed for edge inference.

In another comprehensive review, Fang Liu et al. [57] summarizes the existing edge computing systems and introduces emblematic projects. In their work, the authors contrast edge computing systems and tools like Cloudlet [87], SpanEdge [85], and AirBox [9]; Open Source Edge Computing Projects like CORD [18], Akraino Edge [2], Apache Edgent [3], Azure IoT Edge [7]. Additionally, they review Edge Computing Systems' energy

efficiency and DL optimizations and present critical design issues like multi-user fairness, security, privacy, and cost model. Due to resource constraints, edge inference introduces bandwidth, throughput, power, or efficiency-related challenges. In their work, Alberto Marchisio et al. [63] have examined the challenges above, current trends in hardware accelerators, hardware-level optimizations, run-time optimizations, and software-level optimizations. They discuss the related case studies and present open research challenges in hardware-software co-design, in-memory computing, hardware-aware hyper-parameter tuning, and DNN architectural exploration.

This work presents a comprehensive study of state-of-the-art frameworks, TF-TRT and TFLite, for edge inference. The work evaluates the frameworks on various hardware and DNN-based models to exhibit their effectiveness in optimizing inference on edge devices.

Chapter 3

Framework to Modularize Design Space Exploration

3.1 Primary Contributions

In particular, we make the following contributions:

- We analyze the FAIR data principles and identify ontologies as essential enabling components.
- We create a set of guidelines for designing the HPC ontology, which takes into consideration the special requirements and constraints of making datasets and AI models FAIR in the HPC community.
- A high-level core ontology is designed to provide standard concepts and properties to annotate datasets, and AI models. This is required to facilitate data sharing, searching and ease in benchmarking.
- A set of low-level supplemental components are presented to capture fine-grained information in various subdomains such as computers and performance datasets.
- The proposed work introduces our framework, HPCFair, to enable reusable and reproducible AI models.
- It highlights the capabilities of the proposed framework in pipeline development and design space exploration with a detailed design and API architecture.
- Finally, this work evaluates the functionality with standard AI models and use cases from both HPC and the scientific machine learning communities.

3.2 Motivation

Due to the extreme heterogeneity and complexity of high-performance computing (HPC) node architectures, white-box analytical modeling techniques have become less tractable for analyzing and optimizing large-scale scientific applications. As an alternative, Artificial Intelligence (AI), especially machine learning (ML)-based techniques have been widely

used to address various challenges in HPC, including those related to performance modeling and prediction [62, 94, 90, 60], performance analysis [44, 97, 99, 46, 107], resilience [10, 22], data storage format selection [114], memory optimization [112], scheduling [93], and so on.

However, the HPC community's consensus is that it is difficult to find, access, prepare, share, and reuse high-quality training datasets and AI models, as stated by a recent report of the Department of Energy's Office of Science [30]. This is especially true when ML is applied to analyze and optimize large-scale HPC applications running on heterogeneous node architectures. Researchers spend a significant amount of effort using highly specialized tools (including compilers, performance tools, and runtime systems) to extract and process training datasets from HPC systems. A range of machine learning frameworks is then used to generate AI models. Such datasets and AI models are stored in numerous formats, often without sufficient metadata to describe their semantics. Many datasets and models are underutilized. Part of the reason is that the community has not established standard processes to share the valuable datasets and the corresponding AI models.

As a result, researchers and developers have to resort to costly repeated data collection processes. The HPC community cannot quickly build, evaluate, and reuse machine learning techniques to address pressing HPC challenges.

The problem the HPC community is facing is not unique. Researchers are establishing standard guidelines in many other research communities and recommending best practices to make scientific data Findable, Accessible, Interoperable, and Reusable (FAIR) [109]. Briefly, Findability means that data can be found online, typically through indexing in search engines. Accessibility indicates that data can be retrieved directly or via an approval process. Interoperability means that data follows standards. Finally, reusability denotes that the context of data generated (metadata) is documented so it can be compared to or integrated with other datasets.

Adhering to the previously explained FAIR principles, we proposed a framework, HPCFair, to assist the high performance computing and science communities comprehend the relationship between models, datasets, and data objects. The overarching goal of this framework is to implement FAIR principles for ML-driven HPC. With the APIs provisioned, users can query for datasets and models with metadata, deploy them in their application and run them without the need to worry about the software support and the need to build a model from scratch. It helps to explore models that are trained and tuned for specific tasks; if not available, the user can save them in a central repository for future reuse.

HPCFair empowers researchers to explore the research methodologies, metrics databases, varying datasets, and novel learning techniques. It enables researchers with a framework for pipeline development that refers to codification and automation of stages to produce an AI model. It may consist of multiple sequential steps performing tasks like data loading, preprocessing, model training, including deployment. Also, it renders them a unified platform to optimize the pipeline using tools or compilers like TVM [14] and TensorRT [102]. Notwithstanding the proposed framework's capability to support the generic ML use cases, we primarily focus on tailoring it to suit the large-scale HPC workload.

Additionally, this work focuses on our preliminary work of developing a core component to enable FAIRness of training datasets and AI models for HPC: the HPC ontology, which is a collection of essential concepts and properties capturing information in the domain of HPC, using formal knowledge representation of Web Ontology Language (OWL) [69]. The HPC ontology provides a common vocabulary to describe various datasets and AI

Principle	Description
Findable (F)	F1. Data objects (defined by R1 below) are described with rich metadata F2. Metadata clearly and explicitly include the identifier of the data objects it describes F3. Enable mechanism to find AI models by rich associated metadata F4. Data objects are served in a searchable resource
Accessible (A)	A1. Data objects stored are retrievable by their unique identifier. A2. Communication protocol to retrieve data objects is open, free, and universally implementable. A3. Access to data objects requires authentication and authorization, where necessary. A4. Metadata is accessible even when the data object is no longer available
Interoperable (I)	I1. Data objects use a formal, accessible, and shared language for information description. I2. Data objects are interoperable from one format to another. I3. Data objects include qualified references to other data objects.
Reproducible (R)	R1. Metadata (of the data object) is extensively described with high fidelity. R2. Data objects are served with a public and accessible data usage license. R3. metadata adheres to domain-relevant community requirements.

Table 3.1: FAIR Principles

models. It also provides the semantics to enable the interoperability of heterogeneous data.

3.3 Design Philosophy

In this section, we will be discussing our solution to help enable FAIRify AI models. Henceforth, we will address our proposed approach as "HPCFair" [104]. We first detail how our solution addresses the FAIR principles. Next, we present design architecture and the implementation details of HPCFair. Designing as a three-tier architecture will enable us to implement each component as an independent module with minimal dependencies and is easily extensible to the other language APIs.

We modularize the storage of data objects' metadata enabling efficient findability. The indexed metadata allows users to search for the required data objects based on tags or keywords. We store the metadata in the JSON-LD format to ensure that it can be accessed via open and standard communication protocols like API calls. In addition to models and datasets, we also provide support to store user implemented custom modules in a format that can easily be discovered, loaded, and employed in a pipeline. At the moment, we provide ONNX support to convert the models. We are working towards implementing checkpoint conversions for the selected models. Also, while saving any new object, we check for duplicate insertions based on primary keys, eliminating any redundant information. We currently support access to public data objects and present steps to access any behind the login data objects. Similarly, while loading any data object,

we check for its existence in the cache. In such scenarios, we provide users with either a reuse option or newly force-load the data object. We aim to incorporate authentication checks to ensure that access is granted to only authorized users.

3.3.1 Overview

As shown in Figure 3.1, HPCFair has a front-end connected to several components implementing tags-based search, user notification, load and store of models and datasets. It also contains a supportive component (HPC Ontology [55]) to provide metadata and an advanced component to automatically synthesize workflows. These two components are still under development and are not within the scope of this chapter.

From our detailed analysis of existing state-of-the-art frameworks, we observed that `MLCube`, in its pre-alpha stage of development, offers a perspective that is easily extensible and obeys the "plug-and-play" philosophy. Considering `MLCube` as a basis, we have further implemented enhancements to bridge gaps in achieving FAIR AI for HPC.

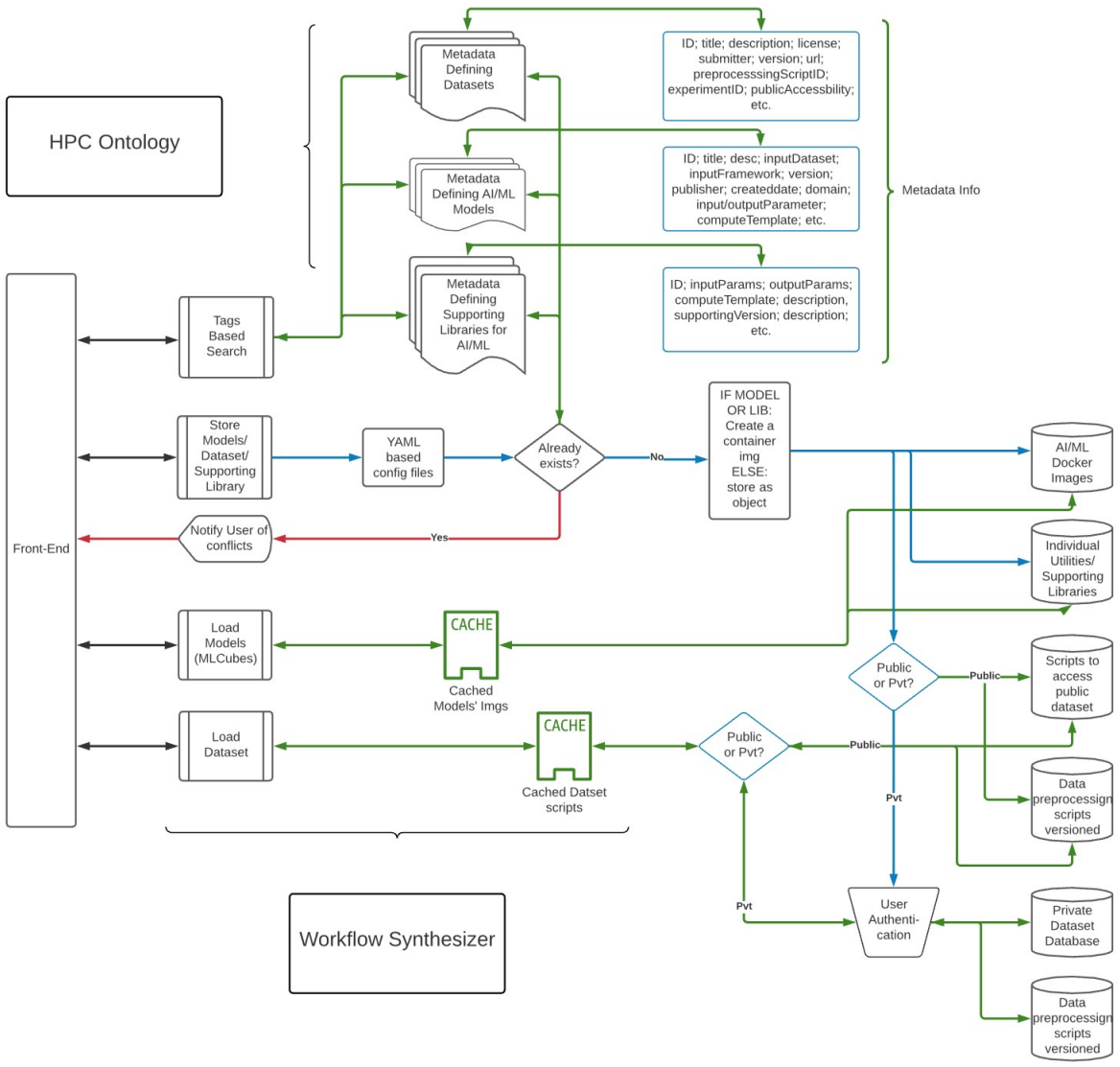


Figure 3.1: HPCFair: The Proposed Architecture

We developed our framework as a Python library for a lightweight implementation. We will extend it to support other languages such as C++, Java, etc. in the future. Developers are provided with CLI support to discover and load the required components. We store detailed information about each component in Github repositories in the JSON-LD format. It helps us store a data object and associated files, thus keeping the relationship among them intact. Also, the JSON-LD format allows avenues to be converted into a more efficient search data structure which is our future direction. The requested information is provided to the developers in the dictionary format (key: value) that is easy to comprehend.

3.3.2 High-Level Core Ontology

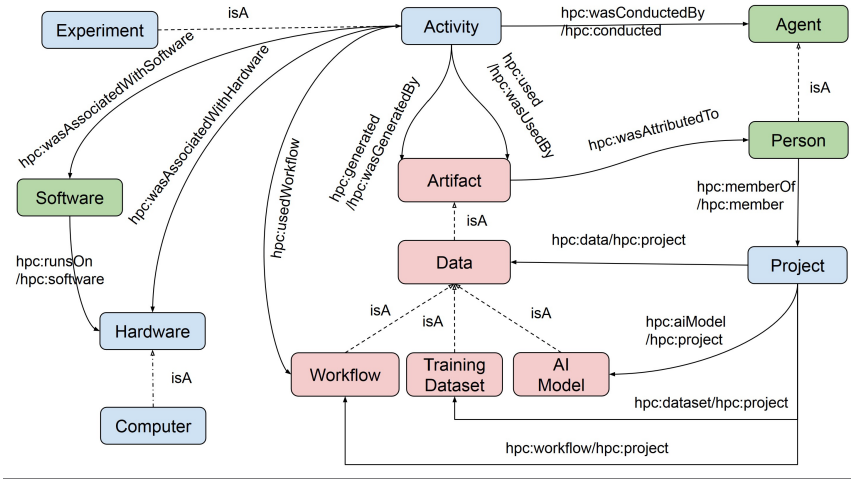


Figure 3.2: Major High-level Concepts and Relationships of the HPC Ontology

In this section, we present representative high-level concepts and their corresponding properties of the HPC ontology.

3.3.3 Basic Scenario and Naming Convention

A basic scenario described by the HPC ontology is that *some people who are members of a project used some software and hardware to conduct some experiments, which in turn used some input data to generate training datasets or AI models.* The semantics can be mapped to three high level concepts, including Agent, Activity, and Artifact. Essentially, *some Agent conducted some Activity which used some Artifact as input and generated some other Artifact.*

As shown in Figure 3.2, we follow a common naming convention when defining vocabularies in the HPC ontology: Singular nouns in CamelCase are used to indicate a Class. Multiword names are written without any spaces but with each word written in uppercase. Relationship (or Property) names start with lowercase letters. For example, *hpc:Project* means a class while *hpc:project* indicates a property that links some data with its associated project.

Dashed arrows in the figure indicate the *isA* relation between a subclass and its superclass. For instance, both training datasets and AI models are kind of data in this context. Solid line arrows indicate other relationships. Some arrows have only a single label to

denote a single relationship. To simplify the diagram, inverse relationships are combined in a single arrow with a pair of relationship labels. For example, the label of the arrow between Person and Project includes both *hpc:memberOf* and *hpc:member*. Not all edges are shown in the figure to avoid a cluttered figure.

3.3.4 Top level Concept: Thing

To provide provenance of all information, the very top level concept in the HPC ontology, *hpc:Thing*, is associated with a set of fundamental properties (listed in Table 3.2) about its unified resource identifier, the type of the ID (such as Open Researcher and Contributor ID and Digital Object Identifier), name, URL, etc. Any other concepts (from both high and low levels) are direct or indirect subclasses of *hpc:Thing*. They naturally inherit all the fundamental properties of *hpc:Thing*. The *hpc:Thing* node and its edges are not shown in Figure 3.2 to simplify the figure.

Property	**Data-type**	**Description**
hpc:id	xsd:anyURI	URI of the thing
hpc:idType	xsd:string	Type of the ID, such as ORCID, DOI, etc.
hpc:name	xsd:string	Name of the thing
hpc:alternateName	xsd:string	An alias for this item
hpc:description	xsd:string	Short description
hpc:url	xsd:anyURI	URL of official website of a thing
hpc:submitter	xsd:anyURI	Who submits this piece of info.
hpc:submitDate	xsd:dateTime	Date of submission

Table 3.2: Properties of the Thing Class

3.3.5 Activity and Experiment

As shown in Figure 3.2, the centerpiece of this design is the concept of Activity, which connects to many other concepts through properties such as *hpc:used*, *hpc:generated*, *hpc:wasAssociatedWithSoftware*, *hpc:usedWorkflow*, *hpc:wasConductedBy*, and so on. An activity is something that occurs over a period of time. An activity could happen after another one, linked using *hpc:wasPrecededBy*. Experiment is a subclass of Activity to represent HPC experiments.

Table 3.3 lists major properties of the Experiment class. For convenience, we also define equivalent properties as needed. For example, *hpc:used* is the same as *hpc:hadInput*. *hpc:wasAWS* is a short name for *hpc:wasAssociatedWithSoftware*. *hpc:wasAWS* is equivalent to *hpc:wasAssociatedWithHardware*. There is also the *wasPrecededBy* property to link a sequence of experiments.

3.3.6 Artifact and Data

We define Data as a subclass of Artifact and further categorize it into Workflow, Training Dataset, and AI model. We have found that in practice, any combinations of mixed scripts,

HPC Ontology Property	Data-type	Description
hpc:used	xsd:anyURI	Input data, ==hpc:hadInput
hpc:generated	xsd:anyURI	Generated data, ==hpc:hadOutput
hpc:usedWorkflow	xsd:anyURI	Workflow used
hpc:wasPrecededBy	xsd:anyURI	Experiments before this one
hpc:wasConductedBy	xsd:anyURI	Link to persons
hpc:wasAWS	xsd:anyURI	Software used
hpc:wasAWH	xsd:anyURI	Hardware used
hpc:startDate	xsd:dateTime	Start time
hpc:endDate	xsd:dateTime	End time
hpc:project	xsd:anyURI	Associated projects

Table 3.3: Major Properties of the Experiment Class

datasets, and AI model files are shared and reused. They are just vaguely categorized as Data currently. In the HPC software analysis and optimization domain, software itself is also a kind of artifact that can be used as input or output data. For example, many compilers, tools and workflows process software as input data to generate program analysis information. So there is a property link (*isA*) between Software and Data also. Again, this edge is not shown in Figure 3.2 to avoid cluttering.

Figure 3.3 shows the partial class hierarchy rooted at Artifact. *AIModel* is equivalent to *MachineLearningModel* in the figure.

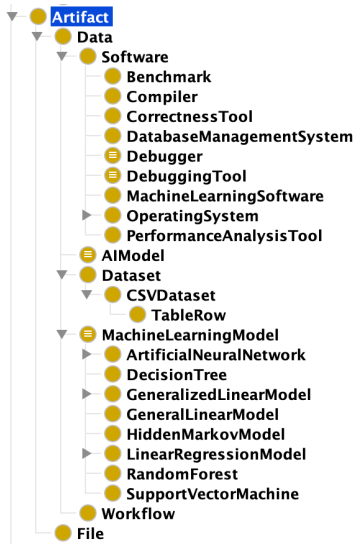


Figure 3.3: Partial class hierarchy of artifact

Table 3.4 shows essential properties for the Data class. There are properties about license, version, associated projects, as well as experiments generating or using the data. Note that Data can be associated with one or more files through the *hpc:file* property. Each file in turn has its properties such as name, size, format, MD5, URL, etc. Data can

be derived from other data in a sequence of experiments. So we have *hpc:wasDerivedFrom* to indicate such a property. In the domain of program analysis and optimization, Data generated often has one or more focus applications or machines. We introduce *targetApplication* and *targetMachine* to directly support such links.

Property	**Datatype**	**Description**
hpc:license	xsd:string	License of the data
hpc:version	xsd:string	Version number
hpc:subject	xsd:string	Performance modeling, optimization...
hpc:file	xsd:anyURI	Associated files
hpc:project	xsd:anyURI	Link to the associated projects
hpc:wasGeneratedBy	xsd:anyURI	output of experiments
hpc:wasUsedBy	xsd:anyURI	Input to experiments
hpc:wasDerivedFrom	xsd:anyURI	Some data was derived from other
hpc:targetApplication	xsd:anyURI	Applications being targeted
hpc:targetMachine	xsd:anyURI	Computers being targeted

Table 3.4: Major Properties of the Data Class

For Dataset and AI Model classes, they inherit all properties of their superclass Data. They also have additional properties as needed. For example, the AI Model class shown in Table 3.5 has extra properties such as source artifacts (*hpc:wasDerivedFromDataset*) used to generate the model, machine learning framework used (*hpc:machineLearningFramework*), configurable parameters (*hpc:params*), accuracy (*hpc:accuracy*), and so on. Note that *hpc:wasDerivedFromDataset* (shown in its abbreviation form *hpc:wasDFD*) is a subclass (or a subproperty in ontology's term) of *hpc:wasDerivedFrom*.

3.3.7 Software

A piece of software (or a program) can be associated with lots of details. We have to narrow down the scope to support generic program analyses and optimizations. In the HPC ontology class hierarchy, Software is a subclass of Data so it has all of Data's properties. We have added a few popular subclasses of Software, including Compiler, OperatingSystem, Benchmark, and so on as shown in Figure 3.3. In addition, we add the properties shown in Table 3.6 to support brief information about a piece of software or benchmark being analyzed or optimized.

3.3.8 Hardware and Computer

The HPC ontology must capture sufficient hardware and computer information to be useful. An HPC computing system can be a single node computer (including a workstation or server) or a cluster with a set of computers. The computer class in the HPC ontology collects the properties describing hardware, performance, and manufacturing related details, as shown in Table 3.7. Note that some properties should link to structured QUDT data types to encode both values and accurate units. For example, the *hpc:harddriveSize*

Property	**Datatype**	**Description**
hpc:modelFormat	xsd:string	Such as protobuf, onnx, h5
hpc:isTunable	xsd:boolean	Tunable during the runtime or not
hpc:params	xsd:string	Configurable parameters
hpc:framework	xsd:anyURI	Such as TF, PyTorch, MXNet, etc.
hpc:modelType	xsd:string	Relevant problem domains
hpc:supportsAccelerator	xsd:boolean	Run on accelerators or not
hpc:accuracy	xsd:double	Accuracy of the model
hpc:overhead	xsd:double	Overhead of the model
hpc:wasDFD	xsd:string	Datasets used to generate this model
hpc:learningType	xsd:string	Supervised, reinforcement learning, etc.
hpc:learningAlgorithm	xsd:string	Decision tree, random forest, etc.
hpc:hyperParameter	xsd:dict	Hyperparameters such as batch size
hpc:modelProperty	xsd:dict	Such as number and types of layers
hpc:inputShape	xsd:string	Input shape to be fed to the model
hpc:inputDatasetFormat	xsd:string	Input format of the dataset

Table 3.5: Major Properties of the AI Model Class

Property	**Datatype**	**Description**
hpc:programLanguage	xsd:string	Programming languages used
hpc:operatingSystem	xsd:string	OS classes supported
hpc:firstReleaseDate	xsd:date	First release date
hpc:latestReleaseDate	xsd:date	Latest release date
hpc:vendor	xsd:string	Link to vendors
hpc:runsOn	xsd:anyURI	Machines runs the software
hpc:input	xsd:anyURI	Input data of the software
hpc:output	xsd:anyURI	Generated output data

Table 3.6: Major Properties of a Software Program/Benchmark

property has *qudt:QuantityValue* to capture sizes in different units such as GigaBytes or TeraBytes.

Property	**Datatype**	**Description**
hpc:vendor	xsd:string	Vendor of the machine
hpc:cpu	xsd:anyURI	Associated CPUs
hpc:cpuCoresPerNode	xsd:integer	CPU core count
hpc:threadsPerNode	xsd:integer	Threads count
hpc:coprocessor	xsd:anyURI	Associated coprocessors
hpc:coprocessorCoresPerNode	xsd:integer	Coprocessor core count
hpc:memorySize	qudt:QuantityValue	Total memory size
hpc:harddriveSize	qudt:QuantityValue	Total hard drive size
hpc:totalPeakPerformance	qudt:QuantityValue	The peak performance
hpc:hasOperatingSystem	xsd:string	The operating system
hpc:hasCompiler	xsd:string	The compilers available
hpc:power	qudt:QuantityValue	Power consumption
hpc:powerEfficiency	qudt:QuantityValue	Such as GFlops/Watts
hpc:hasRmax	qudt:QuantityValue	Obtained peak perf.
hpc:hasRpeak	qudt:QuantityValue	Theoretical peak
hpc:dateCommissioned	xsd:date	Commission date
hpc:site	xsd:string	Hosting facility/institution
hpc:country	xsd:string	Located country

Table 3.7: Major Properties of the Computer Class

We also model CPUs and coprocessors used by a computer. They are linked from a computer object through *hpc:cpu* and *hpc:coprocessor* respectively to provide more information. Table 3.8 shows some basic CPU class's properties. Given the fast changing nature of CPUs and coprocessors (such as Nvidia GPUs), we model their details in low-level supplemental components with vendor-specific terms and properties. We don't put them into the core, high-level ontology.

Cluster is another class in the HPC ontology, inheriting properties from the Computer subclass, to cover additional properties needed for cluster systems. Each cluster object can be linked to one or more computer objects through its *hasNode* property. Table 3.7 shows major properties of the Cluster class. Given the importance of Top500 list, a cluster may have related Top500 ranking information (e.g. *hpc:top500Rank*).

There are a few other high-level concepts such as Person and Project to help describe the semantics of HPC datasets and AI models. We omit their details since their properties are trivial.

Property	Datatype	Description
hpc:processorTech	xsd:string	Codename/model
hpc:processorGeneration	xsd:string	Processor generation
hpc:processorCorePerSocket	xsd:integer	Cores in a socket
hpc:cpuFrequency	qudt:QuantityValue	Frequency
hpc:memoryBandwidth	qudt:QuantityValue	Memory bandwidth
hpc:processorPeakPerformance	qudt:QuantityValue	Processor performance
hpc:vendor	xsd:string	Link to vendors

Table 3.8: Major Properties of the CPU Class

Property	Datatype	Description
hpc:totalClusterCPUCoreCount	xsd:integer	Total CPU cores number
hpc:systemArchitecture	xsd:string	MPP, cluster, etc.)
hpc:computeNodeCount	xsd:integer	Computer node count
hpc:gpuNodeCount	xsd:integer	Number of GPU nodes
hpc:top500Rank	xsd:integer	Top500 ranking
hpc:top500nmax	xsd:integer	Problem size used
hpc:totalCluserMemorySize	qudt:QuantityValue	Total memory
hpc:totalClusterPeakPerformance	qudt:QuantityValue	Total peak performance
hpc:hasNode	xsd:anyURI	Login or compute node

Table 3.9: Major Properties of the Cluster Class

3.3.9 Low-Level Components of Ontology

Low-level components of the HPC Ontology provide fine-granularity concepts and properties to describe subdomains, including hardware details, contents of profiling datasets, and internal details of AI models. They are provided as needed to achieve maximal FAIRness by providing rich attributes to describe data elements. In this chapter, we focus on several example subdomains relevant to the scope of our work.

3.3.10 A Coprocessor: NVIDIA GPU

For heterogeneous architectures, Nvidia's GPUs are popular components of many supercomputers. Six out of the top ten most powerful computers use Nvidia GPUs, as shown in June 2021's Top500 list [98]. We have added a low-level GPU component into HPC ontology to model the properties of GPUs. Similar low-level components can be added in the future to support other types of heterogeneous processors such as TPU, neuromorphic processors, FPGAs and so on.

Table 3.10 shows major properties of a NVIDIA GPU. The properties can be divided into two sets: one is the set of fixed properties of the GPU such as *hpc:theoreticalGPUOccupancy*. The other set includes configurable properties such as *hpc:gpuThreadBlockSize* used to indicate the thread block size configured during a kernel launch.

Property	**Datatype**	**Description**
hpc:dramFrequency	qudt:QuantityValue	Frequency of DRAM
hpc:streamingMultiprocessorFrequency	qudt:QuantityValue	Frequency of Streaming multiprocessor
hpc:activeCyclesOfStreamingMultiprocessor	xsd:integer	Active cycle counts from SM
hpc:theoreticalActiveWarpsPerSM	xsd:integer	Theoretical Active Warps per SM
hpc:theoreticalGPUOccupancy	xsd:double	Theoretical Occupancy
hpc:maxGPUThreadBlockSizeLimitedBySM	xsd:integer	Max block limited by SM
hpc:maxGPUThreadBlockSizeLimitedByRegister	xsd:integer	Max block limited by registers
hpc:maxGPUThreadBlockSizeLimitedBySharedMemory	xsd:integer	Max block limited by shared memory
hpc:maxGPUThreadBlockSizeLimitedByWarps	xsd:integer	Max block limited by warps
hpc:gpuThreadBlockSize	xsd:integer	Launch block size
hpc:gpuThreadGridSize	xsd:integer	Launch grid size
hpc:registersPerThread	xsd:integer	Launch register//thread
hpc:gpuSharedMemoryConfigurationSize	qudt:QuantityValue	Launch shared memory configuration size
hpc:gpuStaticSharedMemorySizePerBlock	qudt:QuantityValue	launch static shared memory
hpc:gpuThreadCount	xsd:integer	GPU thread count
hpc:gpuWavesPerSM	xsd:integer	Launch wave per SM
hpc:gpuUnifiedMemoryRemoteMapSize	qudt:QuantityValue	Unified memory remote map

Table 3.10: Major Properties of a NVIDIA GPU

3.3.11 A GPU Performance Dataset: XPlacer

When building AI models for program analyses and optimizations, performance profiling information is often a critical part of the corresponding training datasets. We select a dataset from XPlacer [112] as an example. The dataset contains GPU performance profiling data used to build models guiding GPU memory placement choices for arrays.

A key observation here is that program optimizations often happen at the kernel (or function) level. So we provide properties associated with kernel performance. Table 3.11 lists properties to capture Nvidia GPU's performance profiling information. The properties can be grouped into several categories, including basic information about a kernel and its major data objects (arrays), execution time, and hardware counter based information such as cache utilization rate, page faults, data transfer sizes. Finally, for optimization problems selecting optimal code variants, we provide *hpc:codeVariant* and *hpc:bestCodeVariant* to annotate labels.

3.3.12 Serving models, datasets, and data objects

Discussing storage of a model, a dataset, or an individual component, we consider storing the AI components and the experiment's metadata and runtime system configuration information. This will help the users to reproduce the results with added correctness. We employ `MLCube` to store the neural network-based models as containerized images. To store the ML models, we *pickle* them, i.e serialize to the disk as an `MLCube`. It offers `mlcube_ccokiecutter` support to readily generate a containerized image given code, data, and docker file. Instead of tightly coupling the dataset and the model, we create a containerized image of the model alone, facilitating the pipeline development. A containerized image consists of YAML-based configuration files with information about the model and any associated components such as runtime libraries. A uniqueness check is performed to ensure that there is no duplicate submission. Understanding that HPC and scientific workloads might be public or restricted in nature, we have authentication mechanism in our plan. The future plan is to authenticate users for access to particular data objects, thus serving private data objects along with public data objects. The authentication and authorization should be performed for all the actions on any restricted data object.

3.3.13 Tags-based search

As shown in Figure 3.1, a user can perform "tags-based search" that inherently retrieves information from the enriched metadata. The metadata information is categorized into incorporating components to support efficient and low-latency fetch. Based on the information presented by the search action, a user can further load discovered datasets, models, or any other components. When a load request is placed, an inspection is performed to verify if that component exists in the cache directory. In that case, a duplicate fetch is avoided by reusing the existing component. A user can still "force" load, deleting the current component and downloading it anew. This is more to provide the caching behavior to reduce the time-to-respond. Further, we store metadata broadly classified as models, datasets, and data objects distinctly from each other. The modularity allows an efficient search and facilitates user queries, create, update, read, and delete (CRUD) operations.

Property	Datatype	Description
Kernel Information		
hpc:kenelName	xsd:string	Kernel/function name
hpc:benchmark	xsd:anyURI	Associated benchmark
hpc:commandLineOption	xsd:string	Command line Options
hpc:arrayName	xsd:string	Name of array
hpc:allocatedDataSize	qudt:QuantityValue	Memory allocation size
hpc:beginMemoryAddress	xsd:string	Beginning array address
hpc:endMemoryAddress	xsd:string	Ending array address
Performance Information		
hpc:cycle	xsd:integer	Profiled cycle count
hpc:executionTime	qudt:QuantityValue	Execution time
hpc:numberOfCalls	xsd:integer	Number of calls
hpc:averageExecutionTime	qudt:QuantityValue	Average execution time
hpc:minExecutionTime	qudt:QuantityValue	Minimum execution time
hpc:maxExecutionTime	qudt:QuantityValue	Maximum execution time
hpc:executionTimePercentage	xsd:double	Percentage of time spent
hpc:memoryThroughputRate	xsd:double	Memory Throughput
hpc:dramUtilizationRate	xsd:double	DRAM utilization rate
hpc:l1CacheUtilizationRate	xsd:double	L1/Tex utilization rate
hpc:l2CacheUtilizationRate	xsd:double	L2 utilization rate
hpc:achievedGPUOccupancy	xsd:double	GPU occupancy
hpc:achievedActiveWarpsPerSM	xsd:integer	Active warps per SM
hpc:cpuPageFault	xsd:integer	CPU page fault count
hpc:gpuPageFault	xsd:integer	GPU page fault count
hpc:hostToDeviceTransferSize	qudt:QuantityValue	HostToDevice transfer
hpc:deviceToHostTransferSize	qudt:QuantityValue	DeviceToHost transfer
Labels		
hpc:codeVariant	xsd:integer	Code variant ID
hpc:labeledCodeVariant	xsd:integer	Best variant ID

Table 3.11: Major Properties of Kernel Performance Profiling Data

3.3.14 Pipeline development support

Researchers often want to compare their results against various hand-tuned libraries or custom modules like cost functions. Also, it is not profitable to re-implement the same algorithm from scratch for the same experiment or create a pipeline. In such scenarios, reusability of data objects to achieve an experiment pipeline is highly critical. We enable developers to serve and load data objects on demand. The versioning of data objects allows having multiple versions of the same data object. A ranking system based on usage makes it possible to rank them in a longer run. The detailed metadata of these data objects permits usability specific to applications. In the subsequent sections, we would discuss the encoding of datasets, models, and individual data objects like ML libraries, workflows (experimentations, scripts, etc.) and the granularity of the metadata information stored.

3.3.15 Metadata

An AI project consists of many components. We have classified these components into the following categories: dataset, models, individual libraries, supporting scripts like pre and post-processing, associated experiments or workflows, and runtime system configuration. As shown in listing 3.5, all information is stored into the JSON-LD format using hierarchical key-value pairs. The keys include standard metadata keywords such as `@id` and `@title`. We also provide additional keys with a prefix of `hpc:`.

For example, every dataset is uniquely identified by its value of the `@id` key. The `hpc:tags` are used to manage information like version, license, and tasks. It is a dynamic field permitting users to add user-defined properties. We further store metadata describing the associated files and workflows or experiments to reproduce the research submissions. Additionally, to support citations and locate relevant publications, we save citations as a linked data field.

```
{
    "@id": "http://example.org/DA000001",
    "@title" : "MNIST",
    "@description" : "The MNIST dataset",
    "hpc:submitter" : "admin",
    "hpc:tags" : ["Version" : 0.1, "License" : "MIT", "tasks" : "img classification"],
    "hpc:associatedFiles" : "pre_process_mnist",
    "hpc:associatedExpt" : "expt_img_classification",
    "hpc:citation" : "@article{lecun2010mnist}",
}
```

Listing 3.1: Selected fields from the dataset metadata

Similarly, for the models as presented in Listing 3.2, we collect the model's submitted format which may be be a `saved_model, onnx,` or `h5` formats, and so on. The `hpc:isTunable` flag informs the user whether the model is tunable during the runtime or not. Furthermore, `hpc:hyperParams` consists of the parameters that can be passed as arguments to the model during runtime. Metadata for machine learning frameworks and model types enables an efficient search and enhances the framework's usefulness. We also present acceleration support and available metrics to the user. It lets the user choose metrics of interest instead of evaluating all.

```
{
    "@id": "http://example.org/MD000001",
    "@title" : "SSD_MobileNet_v2",
    "hpc:modelFormat" : "pb",
    "hpc:isTunable" : "false",
    "hpc:hyperParams" : "",
```

```
    "hpc:machineLearningFramework" : "tensorflow",
    "hpc:modelType" : "SSD",
    "hpc:acceleratorSupport" : "true",
    "hpc:metrics" : "throughput,latency",
    "hpc:tags" : {"category":"object detection", "dataset":"COCO", "License" : "MIT"}
}
```

Listing 3.2: Selected fields from the model metadata

Another significant component is experiment or workflow-related metadata as depicted in Listing 3.3. It describes the files or prerequisite actions required as part of the experiment. The linked workflow files are the command files needed to set up the environment or perform an action. In cases where a whole project package is submitted, a workflow can be leveraged to reproduce the results. To standalone execution, we have fields to record the required software and hardware, in addition to the dataset and model used as part of the experiment. From experience, we have seen that it is critical to have system configuration details to reproduce the expected results. Hence, we provide the functionality to store them.

```
{
    "@id" : "http://example.org/EX000001",
    "@url" : "https://github.com/userX/repoY/XPlacer-Adapter.md"
    "hpc:associatedWorkflow" : "workflow_file_ex000001",
    "hpc:reqSoftwares" : ["scikit-learn","pandas","skl2onnx","onnxruntime","hyperopt"],
    "hpc:reqHardware" : "nvidia-GPU",
    "hpc:sysConfig" : ["OMP_NUM_THREADS":"4","data_format":"NHWC","kmp_affinity":["
    granularity":"fine,compact,1,0"]]
    "hpc:associatedDataset" : ["AWS_data.csv","IBM_data.csv","Merged_data.csv"],
    "hpc:associatedModel" : ["modelLearnerGUI.py","offline_trainer.py"],
}
```

Listing 3.3: Selected fields from the experiment metadata

3.4 Evaluation

This section presents some preliminary use cases using the current draft HPC Ontology, including providing standard metadata for annotating various data and answering questions posed as SPARQL queries.

An AWS machine is used to run experiments. It has 4 cores of Intel Xeon CPU E5-2676 v3 running at 2.40GHz and 16 GB main memory. The evaluation uses a set of tools. Protege v5.5.0 is used for ontology development. Typically raw data is stored in CSV files. Tarql (git hash #b06b4dd) [91] is used to automatically convert CSV files into RDF triples saved into N-Triples (.NT) files. Tarql's conversion rules are manually specified using SPARQL 1.1 syntax. Blazegraph v2.1.6 is used as the graph database supporting standard RDF/SPARQL APIs. All N-Triples files are loaded into a namespace of Blazegraph in a triples mode and inference turned on.

JSON-LD, a JSON-based format storing Linked Data, is used as the main format to present annotated data. JSON-LD can be viewed as JSON plus Context and ID information. The context is used to define the short-hand names. IDs are unique name identifiers to describe keys in the document. These IDs are used as the keys of JSON's key-value pairs to unambiguously encoding information linked to formally defined entities in an ontology. We use a Python script calling RDFLib APIs to generate the JSON-LD output from .NT files.

3.4.1 Providing Metadata for Top500 Supercomputers

Machine properties are fundamental information in HPC. We use the concepts and properties of the HPC ontology to annotate the tabular information of the fastest machine, Supercomputer Fugaku, published in June 2021's Top500 list [98].

The HPC Ontology has all the concepts and properties needed to fully annotate the content of the entire top 500 spreadsheet. Listing 3.4 shows various example properties of Fugaku in JSON-LD format. All properties are stored in key-value pairs. The machine is uniquely identified by its system ID (179807) assigned by the Top500 website. The HPC ontology provides vocabularies of property names, such as *hpc:name*, *hpc:Cluster*, and *hpc:power*. The QUDT ontology provides structured values with detailed information for the units used, such as MegaHZ and KiloWatt. Note that QUDT originally does not have the TeraFLOPS unit, we have extended it to support this.

```
{
  "@id": "https://www.top500.org/system/179807",
  "@type": "hpc:Cluster",
  "hpc:name": "Supercomputer Fugaku",
  "hpc:top500Rank": 1,
  "hpc:vendor": "Fujitsu",
  "hpc:country": "Japan",
  "hpc:cpuArchitecture":"Fujitsu ARM",
  "hpc:cpuFrequency": { "@id": "_:B9e8a3"  },
  "hpc:hasOperatingSystem": "Red Hat Enterprise Linux",
  "hpc:hasRmax": {  "@id": "_:B17f29"  },
  "hpc:power": { "@id": "_:N15c5a" },
  "hpc:processorCorePerSocket": 48,
  "hpc:processorGeneration": "Fujitsu A64FX",
  "hpc:site": "RIKEN Center for Computational Science",
  "hpc:systemArchitecture": "MPP",
  "hpc:systemModel": "Supercomputer Fugaku",
  "hpc:totalClusterCPUCoreCount": 7630848
},

{ "@id": "_:B17f29",
  "@type": "qudt:QuantityValue",
  "qudt:unit": { "@id": "http://qudt.org/vocab/unit/TeraFLOPS"},
  "qudt:value": "442010.00"
},

{ "@id": "_:B9e8a3",
  "@type": "qudt:QuantityValue",
  "qudt:unit": {"@id": "http://qudt.org/vocab/unit/MegaHZ"},
  "qudt:value": "2200"
},

{ "@id": "_:N15c5a",
  "@type": "qudt:QuantityValue",
  "qudt:unit": {"@id": "http://qudt.org/vocab/unit/KiloW"},
  "qudt:value": "29899.23"
}
```

Listing 3.4: Example Fugaku's Information Annotated Using HPC Ontology

3.4.2 Providing Metadata for Datasets and AI Models

We are developing a web portal to allow users to submit information about training datasets and AI models in HPC. To make the submitted data complaint with the FAIR principals, standard metadata must be used to annotate the datasets and AI models. The HPC ontology can serve as the metadata for this purpose.

Listing 3.5 shows a benchmark software package [54] which was used as an input dataset for a code similarity analysis experiment. It is easy to identify metadata for its name, license, subject, related project, and so on.

```
{
  "@id": "http://example.org/dataset/DA000005",
  "@type": "hpc:Dataset",
  "hpc:description": "microbenchmark kernels in C/C++ and Fortran",
  "hpc:hasIDType": "System Generated",
  "hpc:license": "BSD",
  "hpc:name": "DataRaceBench micro-benchmarks",
  "hpc:project": {
    "@id": "http://example.org/project/PR000002"
  },
  "hpc:subject": [
    "OpenMP",
    "Data Race Detection",
    "Computer Science"
  ],
  "hpc:url": "https://github.com/LLNL/dataracebench/tree/master/micro-benchmarks",
  "hpc:version": "1.3.2"
}
```

Listing 3.5: Example Dataset Annotated Using HPC Ontology

Listing 3.6 shows metadata for an AI model released by a study [112]. Besides common metadata for its name and type, it additionally has metadata specific to this AI model, such as *hpc:wasDerivedFromDataset*, *hpc:learningAlgorithm*, and *hpc:targetMachine*.

```
{
  "@id": "http://example.org/AIModel/MO000001",
  "@type": "hpc:AIModel",
  "hpc:contributor":
    {"@id": "http://example.org/person/PI000013"  },
  "hpc:description": "Onnx version of the decision tree model for XPlacer",
  "hpc:wasGeneratedBy":
     {"@id": "http://example.org/experiment/EX000002"},
  "hpc:hasIDType": "System Generated",
  "hpc:wasDerivedFromDataset": [
    {"@id": "http://example.org/dataset/DA000001"},
    {"@id": "http://example.org/dataset/DA000002"},
    {"@id": "http://example.org/dataset/DA000003"},
    {"@id": "http://example.org/dataset/DA000004"}
  ],
  "hpc:isTunable": false,
  "hpc:learningType": "Supervised ",
  "hpc:modelFormat": "ONNX",
  "hpc:modelType": "Prediction",
  "hpc:learningAlgorithm" : "Decision Tree",
  "hpc:name": "decisionTree.onnx",
  "hpc:project":
       {"@id":"http://example.org/project/PR000001"},
  "hpc:subject": [
    "GPGPU",
    "Heterogeneous Systems",
    "Data Placement",
    "Decision Tree"
  ],
  "hpc:supportsAccelerator": true,
  "hpc:targetMachine":
      {"@id": "http://example.org/cluster/lassen"},
  "hpc:url": "https://github.com/xyz/decisionTree.onnx"
  },
  "hpc:version": "1.2.1"
}
```

Listing 3.6: Example AI model annotated using HPC ontology

3.4.3 Encoding GPU Profiling Dataset Stored in a CSV File

Low-level components of the HPC ontology can be used to encode the internal content of datasets or AI models. Listing 3.7 shows the annotation of a row of a CSV file representing a dataset released by a study [112]. Each row represents information about an array

accessed within a kernel code variant and the kernel's associated profiling metrics such as GPU data transferring data sizes and page faults. The dataset was used to train a model deciding which code variant delivers the best performance. Using properties defined in Table 3.11, the annotated json-ld file contains accurate, machine readable information for each cell, which facilitate interoperability and reusability of the dataset.

```
{
  "@id": "https://github.com/xyz/rodinia_3.1_cuda_bfs_datalevel-lassen.csv#L11",
  "@type": "hpc:TableRow",
  "hpc:BeginMemoryAddress": "0x200060080000",
  "hpc:EndMemoryAddress": "0x200060090000",
  "hpc:allocatedDataSize": 65536,
  "hpc:arrayID": "1",
  "hpc:arrayName": "h_graph_mask",
  "hpc:codeVariant": "111100",
  "hpc:commandLineOption": "graph65536",
  "hpc:cpuPageFault": 2,
      "hpc:deviceToHostTransferSize": {
        "@id": "_:Nca485"
      },
      "hpc:hostToDeviceTransferSize": {
        "@id": "_:Na7702" }
    },
  {
   "@id": "_:Na7702",
   "@type": "qudt:QuantityValue",
   "qudt:unit": {
     "@id": "http://qudt.org/vocab/unit/KiloBYTE"   },
   "qudt:value": {
     "@type": "http://www.w3.org/2001/XMLSchema#decimal",
     "@value": "192.0"   }
  },
  {
    "@id": "_:Nca485",
    "@type": "qudt:QuantityValue",
      "qudt:unit": {
        "@id": "http://qudt.org/vocab/unit/KiloBYTE"
      },
   "qudt:value": {
     "@type": "http://www.w3.org/2001/XMLSchema#decimal",
     "@value": "0.0"
   }
  }
```

Listing 3.7: Example profiling dataset annotated using HPC ontology

3.4.4 Enabling Various Queries

A standard approach to evaluating an ontology is to check if the ontology can be used to formulate questions asked by intended users. Such questions often are called competency questions. We anticipate some typical questions from a HPC user may include the following:

- Q1: What are the ids of datasets from a research project named "Xplacer"?
- Q2: What are the names of AI models available for a supercomputer named "lassen"?
- Q3: What AI models are available for machines with GPUs named "Nvidia V100"?
- Q4: What datasets of projects funded by NSF are available for building Performance Prediction models?
- Q5: What workflows are available for generating AI models guiding "Heterogeneous Mapping" of a benchmark (e.g. the NAS Parallel Benchmark or NPB) running on a machine using AMD GPUs?

We populated a Blazegraph RDF database with information encoding a few example datasets and AI models found in the literature, assuring diversified datasets and models across widely researched domains. We gathered data from projects like XPlacer [112] that come with the workflows defining dataset collection and offer models to perform experiments. Further, there were scenarios when researchers publish their results as a package or complete tool. The ProGraML [**cummins2021a**] and MLGO [**trofin2021mlgo**] are such considered examples. Additionally, we included literature from HPC Energy Research [80] providing datasets or models alone. We also collected some datasets and refined models hosted at Kaggle. They are related to the GPU Kernel Performance [8, 74, 89].

Listing 3.8 shows the SPARQL queries sent to the Blazegraph database and the corresponding answers obtained, for Q1 through Q3. Listing 3.9 shows queries and results for Q3 and Q4. The results show that HPC Ontology is complete to support the concepts and properties expressed in these queries which in turn obtained desired results.

```
PREFIX hpc: <https://example.org/HPC-Ontology#>
# Query for Q1: dataset ids of a project
#-------------------------------------------------------
SELECT ?ds
WHERE { ?pid rdf:type hpc:Project .
        ?pid hpc:name "Xplacer" .
        ?pid hpc:dataset ?ds }

# Results: IDs of datasets
# <http://example.org/dataset/DA000001>
# <http://example.org/dataset/DA000002>
# <http://example.org/dataset/DA000003>
# <http://example.org/dataset/DA000004>

# Query for Q2: AI models' names of a supercomputer
#-------------------------------------------------------
SELECT ?model_name
WHERE { ?model_id rdf:type hpc:AIModel .
        ?model_id hpc:targetMachine
            <http://example.org/cluster/lassen> .
        ?model_id hpc:name ?model_name }

# Results : names of AI models
# decisionTree.onnx
# randomForest.onnx

# Query for Q3: machines with NVidia V100 and their models
#-------------------------------------------------------
SELECT ?machine ?model_id
  WHERE { ?gpu rdf:type hpc:GPU .
          ?gpu hpc:name "Nvidia V100".
            ?machine hpc:coprocessorModel ?gpu .
            ?model_id hpc:targetMachine ?machine .
            ?model_id rdf:type hpc:AIModel .
            ?model_id hpc:name ?model_name  }

# Results: machine names and AI model IDs
<http://example.org/cluster/lassen> <http://example.org/AIModel/MO000001>
<http://example.org/cluster/lassen> <http://example.org/AIModel/MO000002>
```

Listing 3.8: SPARQL queries to answer competency questions 1-3

```
PREFIX hpc: <https://example.org/HPC-Ontology#>

# Query for Q4: NSF project's datasets for building performance prediction models
#-------------------------------------------------------
SELECT ?project_id ?ds_id
WHERE {
        ?project_id rdf:type hpc:Project .
        ?project_id hpc:fundedBy "the National Science Foundation" .
        ?ds_id hpc:project ?project_id .
        ?ds_id rdf:type hpc:Dataset .
        ?ds_id hpc:subject "Performance Prediction" }
```

```
12
13 # Results: project IDs and dataset IDs
14 <http://example.org/project/PR000003> <http://example.org/dataset/DA000007>
15 <http://example.org/project/PR000003> <http://example.org/dataset/DA000008>
16 <http://example.org/project/PR000003> <http://example.org/dataset/DA000009>
17
18 # Query of Q5: workflow generating AI models for NPB's heterogeneous mapping on AMD GPU
19 SELECT ?pid ?pname
20 WHERE {
21         ?pid rdf:type hpc:Workflow .
22         ?pid hpc:subject "Heterogeneous Mapping" .
23         ?pid hpc:name ?pname .
24
25         ?pid hpc:targetMachine ?machine_id .
26         ?machine_id hpc:coProcessor ?gpu_id .
27         ?gpu_id hpc:vendor "AMD" .
28
29         ?pid hpc:targetApplication ?app_id .
30         ?app_id hpc:name "NPB" .
31       }
32
33 # Results
34 <http://example.org/workflow/WF000020>  OpenCL Heterogeneous Mapping
```

Listing 3.9: SPARQL queries to answer competency questions 4-5

To evaluate HPCFair for its compliance with FAIR principles, we have conducted the following diversified experiments. We have compared the results for correctness in regards to the original experiments. Additionally, we have demonstrated how the data objects' reusability and interoperability can further tune a model and achieve an efficient software-hardware co-design system.

3.4.5 Evaluating Support for DNN models

To assess the HPCFair framework upon a fundamental use case, we consider experimenting with the MNIST dataset [68]. The MNIST dataset consists of 60,000 examples of handwritten digits forming a test set. The intention here is to replicate the experimentation by MLCube [2], where they employ a simple neural network to classify the earlier mentioned training set into ten classes. In the original approach, the download and train task is unified. In contrast, as shown in Listing 3.10, we have decoupled them into two independent tasks to 1) reuse the dataset, line 18 and 2) train the model, line 23. In the background, we use runners provided by MLCube to run cubes on different platforms, including docker and singularity. The parameter.yaml file contains the hyperparameters that can be set during the task execution. If not provided during the runtime, the default hyperparameter values are used. As shown in lines 1-2 in Listing 3.10, users can load the `hpcfair_dataset` and `hpcfair_model` as modules. They can search for the MNIST dataset and related models, as shown in lines 5 and 9. Further, the dataset and model can be loaded, and the training task is run with the custom hyperparameters, as shown in lines 18 and 7 respectively.

```
1 >>> from hpcfair import hpcfair_model as model
2 >>> from hpcfair import hpcfair_dataset as dataset
3 >>>
4 >>>
5 >>> model.search("mnist")
6 ['mnist_model_us1']
7 >>> model_mnist = model.load("mnist_model_us1")
8 >>>
9 >>> dataset.get_metadata("mnist")
10 [hpcfair_dataset.DatasetMetadata(
```

[2] https://github.com/mlcommons/mlcube_examples/tree/master/mnist

```
        title='mnist',
        description='The MNIST dataset consists of 70,000 28x28 black-and-white images in
    10 classes (one for each digits), with 7,000images per class. There are 60,000
    training images and 10,000 test images.',
        files=None,
        supporting_files=pre_processing_mnist_0.py,
        isTunable='true'
)]
>>>
>>> dataset.load_dataset("mnist")
"D:\\hpcfair\\data\\mnist.npz"
>>> dataset.apply("mnist","D:\\hpcfair\\data\\","pre_processing_mnist_0",num_threads="4")
Successfully processed the dataset:  python3 pre_processing_mnist_0.py --data=mnist.npz
    num_threads=4
>>>
>>> model.run("train", model_mnist, "D:\\hpcfair\\data\\mnist.npz", log_dir="D:\\logs")
Model successfully trained. Check logs directory for more details.
```

Listing 3.10: Support for generic DNN-models

3.4.6 Evaluating Support for ML libraries

In another evaluation, we applied some machine learning algorithms, such as linear regression and logistic regression, on the GPU runtime dataset [88] as independent data objects. We performed a gradient-descent with batch updates to predict GPU computation time. The dataset includes 14 independent features and 241,600 rows. In this evaluation, we applied data processing files on the dataset stored as dataset metadata. The hyperparameters experimented with are learning rates and convergence threshold.

- Search for the dataset
- Load the dataset
- Apply preprocessing steps as a script
- Search the ML libraries stored as data objects in the pickle format
- Deserialize the ML libraries
- Provide hyperparameter values and train the pipeline; experiment with multiple values
- Use the above model to predict GPU runtime based on varying selected features.

Consequently, the resultant pipeline can be saved as a readily reproducible pipeline, a data object, complying with the FAIR principles.

3.4.7 Evaluating Reproducibility of Published Research

With proliferating research involving ML and DL, it is essential to reproduce presented results with the least effort. Effectuating these needs, we sought to produce the results from the Best Paper Award winner submitted in PACT'17 by Chris Cummins et al. [20]. This work introduces a novel framework DeepTune, proposing heuristics predicting optimal mapping for heterogeneous parallelism and GPU thread coarsening factors using LSTM - a deep neural network.

The artifact submitted tightly depends on the CLgen [16] version and has recommended Ubuntu and Python versions. The containerized packaging of the HPCFair

facilitates the reproduction of the results with efficiency, managing the installation of required dependencies. Also, results achieved from the DeepTune have been confronted with results from the state-of-the-art frameworks from Grewe et al. [35] and Magni et al. [61].

3.4.8 Evaluating Support For Workflows

Ensuing is the evaluation revealing the support for the workflows. Often, there are research submissions in the form of packages where a sequence of steps needs to be performed to set up the prerequisites for reproducibility. One such example is where we evaluated HPCFair for its efficacy to reproduce the results from Xplacer by Xu et al. [112].

```
1  >>> from hpcfair import hpcfair_model as model
2  >>> from hpcfair import hpcfair_dataset as dataset
3  >>>
4  >>> model.search_workflows("xplacer")
5  ['XPlacer']
6  >>> model.get_metadata("xplacer")
7  [hpcfair_model.ModelMetadata(
8          title='XPlacer: A framework for Guiding Optimal Use of GPU Unified Memory',
9          description='The goal is to decide which memory placement policy is best for a
       given data object...',
10         files=None,
11         supporting_files=['xplacer_wf1.sh','xplacer_wf2.sh','xplacer_wf3.sh'],
12         isTunable='true',
13         stepsToRun='xplacerRunFile.txt'
14 )]
```

Listing 3.11: Support for workflow

As prerequisites, the experiment includes steps like building the adapter, collecting the data-level and kernel-level baseline data, merging two levels of baseline data, labeling the merged data, and executing all variants to decide the best-performing variants. The authors do provide a sequence of steps and scripts to be executed to achieve the above. As a solution, we transform them into workflow scripts and serve the related details as metadata in our database. These data objects are bundled together as part of the containerized image. As shown in the Listing 3.11, line 4-6, a user can search for an experiment and associated workflow scripts. Additionally, a user can view the brief description explaining the steps to execute when projects are submitted as packages using the corresponding metadata key.

3.4.9 Evaluating Support For Design Space Exploration

Lately, edge computing has received considerable attention addressing the demand for faster Deep Learning applications at edge devices. This has led to the development of custom accelerators (TPU, GPU, FPGA), DNN-compilers (TVM, TF-Lite), and frameworks (MxNet, Pytorch, TF). However, recognizing a set of components briefed above best suited for a DL task is strenuous. Hence, a comprehensive framework capable of tackling this issue is of paramount relevance for the researchers. In a similar attempt, we have evaluated HPCFair to reproduce the results [105] explaining its capabilities for an efficient Design Space Exploration. A code snippet for the same is presented in the Listing 3.12.

```
1  >>> from hpcfair import hpcfair_model as model
2  >>> from hpcfair import hpcfair_dataset as dataset
3  >>>
4  >>> model.search("resnet")
5  ['ResNet50_v2']
```

```
>>> model_resnet50 = model.load("ResNet50_v2")
>>> data = dataset.load_dataset("img0.jpg", fromPath="D:\\data")
>>>
>>> res_trt = model.run("inference", model_resnet50, data, config="trt", log_dir="D:\\
    logs")
>>> res_tflite = model.run("inference", model_resnet50, data, config="tflite", log_dir="D
    :\\logs")
```

Listing 3.12: Support for design space exploration

The evaluation includes inference using the Resnet50 v2 model trained on the ImageNet dataset [43]. We make the inference pipeline efficient using optimizations proposed by TFLite [34] and TF-TensorRT integrated solutions [76]. The intention is to select the optimal set of components as a pipeline, essentially improving the inference efficiency based on metrics like power consumption, throughput, and reduction in the model size. HPCFair offers models, datasets, and frameworks as data object readily integrable into the pipeline rather than building from scratch. Therefore, instead of repeating from scratch, we adopted the plug-and-play methodology, comparing contrasting design space alternatives fairly for each combination of the framework, accelerator, and optimization engine.

3.4.10 Enabling FAIR Principles by HPCFAIR

We achieve FAIR guidelines to the AI applications and data objects as listed below.

- **Findable:** The metadata associated with data is registered and indexed in a searchable resource. The metadata is assigned a globally unique and persistent identifier. This enriched metadata enhances searchability. A user will be able to search for the components corresponding to the application's requirements.
- **Accessible:** The metadata is retrieved using a standardized communication protocol. It enables users to publish or discover their AI components efficiently. Further, access to the metadata is authorized and authenticated wherever necessary.
- **Interoperable:** To manage interoperability, we have represented the metadata using a formal language, JSON-LD [92]. We support qualified references among the stored metadata and data objects. Additionally, to maintain interoperability at the application level, we serve metadata information concerning individual supporting files associated with the data object. We also support ONNX [77], equipping application users to transform models from one format to another.
- **Reusable:** The scientific community oftentimes interacts among researchers to share and reuse crucial components. We provide metadata with detailed provenance to reuse the components to build an AI pipeline by plugging the data objects. The loosely coupled nature of the stored data enables efficient development.

3.5 Use Cases Reinforcing Scientific Machine Learning Applications

The purpose of HPCFair is not limited to the assessed experiments and features' support. Subsequent use cases show how it can be used to apply FAIR principles to the scientific community applications.

3.5.1 Democratizing Datasets and Models in Medical Research

Part of the Cancer Distributed Learning Environment (CANDLE) project, Uno [100] is a cancer deep learning benchmark to predict drug response based on molecular features of tumor cells and drug descriptors. The training task on all data sources is a slow process. But there are hand-tuned configurations that can speed up the process for a single data source. The training and inference can further be optimized using a pre-staged dataset. This requires regeneration of the dataset for varying configurations followed by data processing and many other significant hyperparameters like `batch_size, cache, use_landmark_genes, preprocess_rnaseq, no_feature_source, shuffle`, etc. The HPCFair enables democratization by offering datasets and preprocessing scripts as data objects. The access to the data objects is authenticated to allow for only approved users to access them.

Further, the hyperparameters can be set using a YAML-based configuration file information stored as metadata. A trained model can be served to be reused later for inference. This also would help with easy and efficient porting of the models to diverse HPC systems such as ThetaGPU, Polaris at ALCF, Summit, Frontier at OLCF and Cori, PerlMutter at NERSC.

3.5.2 Predicting Cosmological Parameters Efficiently

CosmoFlow [66] is a scalable TensorFlow-based deep learning framework to process sizeable 3D cosmology datasets on a modern HPC platform. The model aims to predict a couple of parameters from the distribution of matter. The dataset consists of simulation boxes of dark matter distribution. It is further augmented and pre-processed. HPCFair stores the script that converts the original input data from `.npy` to `.tfrecord`. Additionally, it stores hyperparameters in a script. The processed data can be stored as a versioned data object to account for the efforts required to regenerate the dataset.

3.6 Discussion

With the proliferating AI research and development among the HPC and scientific community, the need for a platform to contain data objects in an easily findable, accessible format, enabling interconvertibility and reusability among various frameworks, is indispensable. HPCFair is an attempt in that direction. We have endeavored to abridge gaps in different state-of-the-art frameworks. We exhibited how HPCFair can assist development from a baseline DNN experiment to individual ML Libraries as reusable components, extending to reproduce research results and pipeline development. We also demonstrated how it could be leveraged to expedite and ease the design-space exploration.

HPCFair is our very first step in applying FAIR principles to HPC applications. Where we support only TF-ONNX interconversion, PyTorch-ONNX conversion at present, the eventual intention is to implement checkpointing conversion-like methodology. It will equip the user with the ability to make TF-PyTorch models interoperable. We also aim to have GUI providing better search capabilities. Along with these, we are constantly refining and implementing all the proposed features to proffer the scientific community a unified platform that will make the emerging workloads efficiently reusable and portable. The current implementation of HPCFair will be released as open source software on github once we get necessary approvals from our organizations.

3.7 Related Work

The advancement to promote FAIR principles in the AI universe has burgeoned in the recent past. We extensively studied the existing state-of-the-art frameworks and have identified and overcome any gaps in our proposed framework. We first briefly describe them , compare their features in Table 3.12 and provide our analysis in Section 3.7.9.

3.7.1 The FAIR Data Principles

Accelerating scientific discoveries and innovations depend on a good ecosystem managing experiment data for various purposes such as validation and reusing. Unfortunately, existing scholar publication systems focus mostly on papers. The associated experiment data is either discarded or treated as a secondary citizen with lower standards for publishing. In response to the urgent need in the scientific community to improve the infrastructure supporting the reuse of scholarly data, the Future of Research Communications and e-Scholarship (FORCE11) proposed the FAIR data principles [109] describing four fundamental principles: Findability, Accessibility, Interoperability, and Reusability. These principles are further refined, as shown in Table 3.1.

One of the main goals of the FAIR principles is to achieve machine-actionability in order to process the large amount of scholar data generated daily. This means that data management systems should provide rich and standard information such as the type of digital objects, their formats, licensing, and appropriate operations on them. To be practical, both contextual metadata surrounding a digital object and the content of the digital object should use controlled vocabularies, which in turn are associated with controlled semantics. As a result, several refined FAIR principles shown in Table 3.1 explicitly mention vocabularies (I2), knowledge representation (I1), and community standards (R1.3) to improve machine-actionability.

3.7.2 Ontologies

Ontologies can provide the much-needed controlled vocabularies, knowledge representation, and standards to implement the FAIR data principles. Ontology [101] is a concept originating in Philosophy, referring to the study of the nature of being, as well as the basic categories of them and their relations. In recent decades, it has become a formal way to explicitly represent knowledge in a domain. An ontology [**staab2013handbook**, 101] is a formal specification for explicitly representing knowledge about types, properties, and interrelationships of the entities in a domain. It provides a common vocabulary to represent and share domain concepts. Compared to tree-like taxonomy solely modeling the generalization-specialization relation, an ontology can form a much more complex graph with edges to model any kinds of relationships between entities (represented as graph nodes). Such graphs are often called knowledge graphs in many communities.

Figure 3.4 illustrates an example ontology for Robotics. Each node in the graph represents a concept or instance, and each edge carries a property indicating the relations between the two nodes it connects. For example, there are two nodes named “Agent” and “Object” indicating two concepts (or classes). The “is-a” edge between them means that one is a subclass of the other. An edge labeled ‘instanceOf” denotes that a node is an instance of a node representing a class. Another edge (labeled “hasPart”) between “Robot” and “Arm” indicates that the latter is a part of the former.

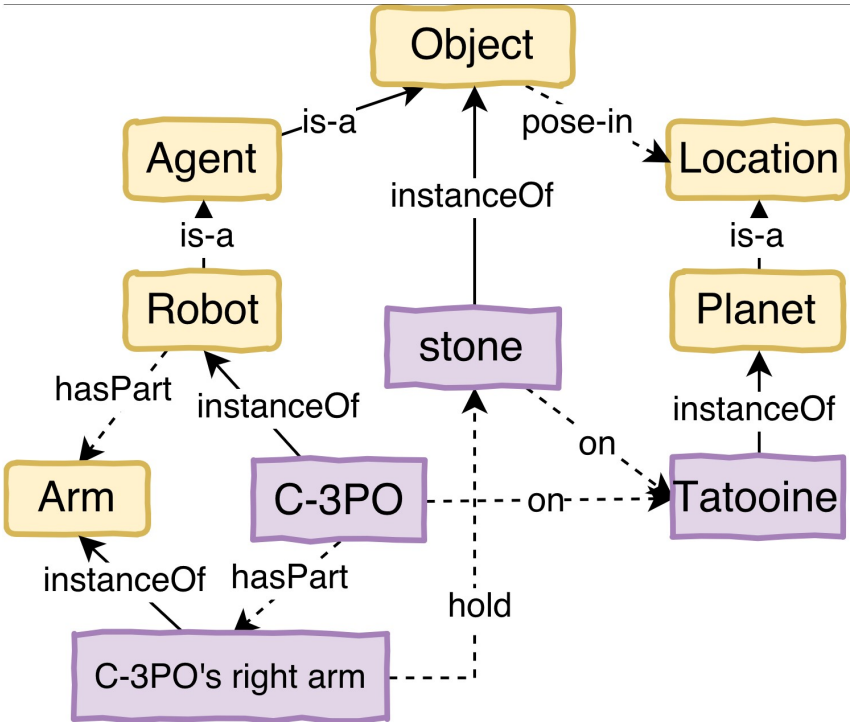


Figure 3.4: An example ontology on robotics.

A formal language for expressing ontologies is the Web Ontology Language (OWL) [69]. OWL is based on the description logic (DL) and is expressive enough for building sophisticated knowledge bases while still supporting efficient inference. Resource Description Framework (RDF) is a fundamental format used to store a wide range of information, including ontologies written in OWL. Each piece of knowledge stored in RDF is represented as a triple, *(subject, property, object)*. For instance, *("Tatooine", instanceOf, "Planet")* states that *Tatooine* is an instance of *Planet*.

An ontology can be queried using a standard RDF query language named SPARQL. An example query in Listing 3.13 can be used to find all robots on the Tatooine planet using the Robotics ontology through the join of two properties. *rdf:type* is a standard RDF property to link a resource as an instance to its class. *:on* is the property to link a resource to its location.

```
SELECT ?s
WHERE { ?s rdf:type :Robot .
        ?s :on :Tatooine }
```

Listing 3.13: Example SPARQL Query on the Robotics Ontology

Numerous ontologies or controlled vocabularies have been developed to enhance interoperability and reusability of data in different domains. For example, EDAM [**ison2013edam**] is an ontology for bioinformatics operations and data types. Schema.org [**guha2016schema**] is designed to improve the interoperability of web data. The Genomic Data Commons of National Cancer Institute [**jensen2017nci**] maintains standard terms and references to public ontologies and vocabularies in dictionary files to share linked clinical and genomic data. Brick [**balaji2018brick**] is an ontology to capture the common terms and relations in the domain of smart buildings.

In summary, the use of ontologies ensures "I"nteroperability and "R"eusability of the FAIR principles. In the context of making HPC training datasets and AI models FAIR, a supportive ontology is a natural prerequisite to provide a standard vocabulary of the

HPC domain. However, there is a lack of effort of creating ontologies for the domain of machine learning-based program analysis and optimizations in high-performance computing (HPC). The focus of this work is to contribute to a design of such an ontology, namely HPC ontology.

3.7.3 MLCube

`MLCube` [67] by MLCommons™ is a containerized interface to machine learning models and datasets. It provides open-source runners capable of running on local machines, cloud servers, or Kubernetes clusters. Being in its infancy, `MLCube` supports tasks like dataset download and training. Users can also create containerized images, for their models, provided they have the dataset, code, and docker files. The generated MLCube can be configured using `mlcube_cookiecutter` APIs. It provides `YAML`-based configuration files that support defining tasks and additional hyperparameter options as runtime flags.

3.7.4 DLHub

Data and Learning Hub for Science (`DLHub`) [13] is a cloud-hosted learning system designed to enable the publication of models with descriptive metadata, persistent identifiers, and flexible access control. It packages models into portable containers, enabling low-latency, distributed serving of these models on various heterogeneous platforms. It implements command-line interface (CLI) and software development kit (SDK) support to store, discover, and publish models. `DLHub` provides optimizations as batching and memoization that enhance the inference performance. To assure that all operations are performed by authenticated and authorized users, `DLHub` utilizes Globus authentication mechanism.

3.7.5 Collective Knowledge

Collective Knowledge Framework (`CK` or `cKnowledge`) [17] provides unified APIs, command-line interfaces, meta-descriptions, and general automation actions to organize and manage research projects as a database of AI components. The customizable program pipeline with software detection plugins and the automatic installation of missing packages enables AI components like models, datasets, compilers, and tools from varying vendors to build and run on unalike platforms. The modular `CK` approach has successfully automated benchmarking, auto-tuning, and co-designing software and hardware for AI. It is also being used to reproduce results from top-tier conferences such as ASPLOS, CGO, and Supercomputing.

3.7.6 MLflow

`MLflow` [113] is an MLOps platform intended to streamline machine learning development, experimentation, and productization. Classified into several autonomous components such as tracking, projects, models, and registry, it can be collectively employed to log runtime information, package supporting tools, and provide a centralized store and APIs to manage the entire lifecycle of `MLflow` models. The lightweight APIs offered by `MLflow` can be utilized with any existing machine learning application or libraries like TensorFlow, PyTorch, and XGBoost. It presents support for notebooks, standalone applications, and the cloud, along with Docker containers.

Category	Feature	Supported Data Object	MLCube	DLHub	CK	MLflow	HuggingFace	TFHub	TorchHub	HPCFair
Findable	Search Capability	Dataset	NA	NA	NA	NA	Yes	Yes	NA	Yes
		Model	NA	Yes	NA	Yes	Yes	Yes	Yes	Yes
	Metadata Information	All	NA	Yes	Yes	Only Models	Only Datasets and Models	NA	NA	Yes
Accessible	Accessibility Options	Dataset	API	API	API/CLI	NA	GUI/API	GUI/API	NA	API
		Model	API	GUI/API	API/CLI	API/CLI	GUI/API	GUI/API	GUI/API	API
	APIs Support	All	NA	Python	Python, JSON	Python, R, Java, REST	Python	Python	Python	Python
Interoperable	Format Conversion	Dataset	NA	NA	NA	NA	Yes	NA	NA	Ongoing
		Model	NA	NA	NA	NA	Yes	NA	NA	Yes
	Domain Support	All	Generic	Generic and Scientific	Generic	Generic	NLP	Generic	Generic	Generic, HPC and Scientific
Reproducible	Offering as Individual Component	All	NA	Limited	Limited	Limited	Only Datasets and Models	Only Datasets and Models	Only Models	Yes
	Train and Inference Support	Model	Train	Inference	Both	Both	Both	Both	Both	Both
	Pipeline Development Support	All	NA	NA	Yes	Limited	Yes	Yes	Limited	Yes

*NA: Not Applicable; GUI: Graphical User Interface; API: Application Program Interface; CLI: Command Line Interface; All: model, dataset, custom ML libraries; limited: not applicable to all the data objects; NLP: Natural Language Processing; Generic: industrial ML applications

Table 3.12: Comparison of the existing state-of-the-art frameworks

3.7.7 Hugging Face

`Hugging Face` [41] offers Transformer models and datasets as open-source libraries that equip developers with abstraction layers to store, load, and distribute pre-trained NLP models like BERT. The Transformer API is powered by transformer architecture that scales with training data and model size, facilitates efficient parallel training, and captures long-range sequence features. It contains over 2000 pre-trained and fine-tuned models. The hub offers a user interface and command line features to load and upload models with associated metadata.

3.7.8 Tensorflow and PyTorch Hub

Tensorflow Hub (`TFHub`) [95] is a platform for distributing, discovering, and reusing machine learning components as self-contained Python modules in TensorFlow (TF). A module and pre-trained weights can be reused to retrain across other related and similar tasks assisting transfer learning. `TFHub` provides modules in various domains like text, video, image, etc., in formats like saved models, TF.js, TFLite, and Coral. Once exported to the disk, the modules are self-contained and can be used as an interface to preprocess the user input. The modules are applied to build the part of the TF Graph.

`PyTorch Hub` [82] is an API and workflow employed to publish pre-trained models to a GitHub repository by adding a Python script that contains functions to load a pre-trained model. These functions, alias "entry points," define a model's input and output. Like other AI model hubs, PyTorch advances research within the machine learning community by allowing researchers and developers to leverage plug-and-play models. It provides an interface to load models and pre-trained weights. As of today, according to the `PyTorch Hub's` official GitHub repository doesn't support hosting pre-trained weights. The users with pre-trained weights need to host them correctly themselves.

3.7.9 Comparing State-of-the-art Frameworks

Table 3.12 compares different state-of-the-art frameworks proposed recently to implement a resolution to facilitate FAIR AI, as briefly described above. We analyzed them to understand support for multiple features and have attempted to distinguish the uniqueness amongst each.

While frameworks like `TFHub` and `Hugging Face` offer search capabilities for datasets and models, none other offer such capabilities for datasets. Other frameworks currently do not support model interoperability except for `Hugging Face`. This is a barrier when researchers want to compare the performance of various frameworks on heterogeneous backends. Also, offering each AI component as a data object is critical. While `CK` does offer support for packaging and reproducing the full result, individual component reuse in pipeline support is limited. `TFHub` and `Hugging Face` offer this support only for the datasets and the models, not custom libraries. Whereas `PyTorchHub` supports only models. The innate nature of `Hugging Face` means that it is extensively built for models used in applications from the natural language processing (NLP) domain. This prevents it from being portable across other disciplines, such as HPC and scientific applications. Among all the existing frameworks, `DLHub` alone supports scientific workloads explicitly. Similarly, `TFHub` or `PyTorchHub` are specific to the TensorFlow-based or Torch-based models. In addition to the discussed frameworks, we also studied various publicly available model zoos from GluonCV [33], Caffe [12], and ONNX [78]. However, most lacked support for datasets and data objects, maintaining only models. Such constraints propose a need for a more generic platform assisting researchers, especially in the HPC and machine learning communities.

Our framework HPCFair aims to address the limitations above: provide support for model interoperability, search capabilities for datasets and models, packaged to run seamlessly, and integrate into any application while catering to HPC and science domains.

Chapter 4

Neural Architecture-Aware Optimizations to Reduce Compilation Time

4.1 Primary Contributions

In this work [106] , we study diverse DNNs and present the effect of neural architecture-aware selection of passes and execution order resulting in efficient lower-level code generation. We evaluate the experimental results on execution time, throughput, GPU utilization, memory, and energy consumption metrics. The main contributions of this work are summarized as follows:

- This work underscores the relevance of neural architecture-aware selection of passes in a DL compiler.
- It evaluates the proposed method against standard optimization level and randomized selection of passes.
- Lastly, we demonstrate how the proposed approach can prune the search space for optimizations selection substantiated with critical metrics.

4.2 Motivation

Deep Neural Networks (DNN) form the basis for many existing and emerging applications. Many DL compilers analyze the computation graphs and apply various optimizations at different stages. These high-level optimizations are applied using compiler passes before feeding the resultant computation graph for low-level and hardware-specific optimizations. With advancements in DNN architectures and backend hardware, the search space of compiler optimizations has grown manifolds. Also, the inclusion of passes without the knowledge of the computation graph leads to increased execution time with a slight influence on the intermediate representation.

The researchers soon identified that the DNN execution differs from the execution of standard computer programs. The network architecture allows extracting parallelism and applying various high-level compiler optimizations specific to tensor operations and particular backend hardware support to these tensor operations. The selection of these compiler optimizations is known as pass selection. The pass selection problem has been

researched over decades for traditional computer programs [72, 40, 70, 71, 64, 65]. Many of the DL compilers like XLA [84], TVM [14], Glow [83], and TensorRT [76] apply a predefined set of high-level optimization passes on a given input computation graph oblivion to the neural architecture. The optimization search space has exponentially expanded with the increase in custom optimization passes and the evolution of neural network architectures. This explosion in the search space limits the use of static rule-based optimization level selection or the application of machine learning techniques to select the best passes.

4.3 Methodology

We started by analyzing the architectural differences in the deep neural networks. Further, we explored various compiler passes available in TVM and their impact on a given neural architecture. Subsequently, we applied them to the deep learning workloads, improving the throughput, latency, and overall performance.

4.3.1 Neural Architecture Analysis

We considered four different classes of neural networks - ResNet [37], MobileNet [86], Bidirectional Encoder Representations from Transformers (BERT) [26], and Single Shot MultiBox Detector-based (SSD) [58] architectures, as summarized in Table 4.3. We focused on image classification, object detection, and natural language processing (NLP) task corresponding to Question Answering. Moreover, we assessed the computation graph in TensorFlow, PyTorch, and MXNet to generalize the applicability of our methodology. The precision mode used is FP32 as quantization is not well supported for different networks in TVM. We evaluted a trained network on the same and different dataset to validate the proposition for correctness.

ResNet50:

The ResNet is a 50-layer deep convolutional neural network (CNN). To address the accuracy saturation and further degradation problem with increasing depth, it uses a deep residual learning framework. The baseline plain network consists of convolutional layers with a global average pooling layer and a fully connected (FC) layer with a softmax activation function in the end. It passes through phases performing the convolution with stride 2, batch normalization, and ReLU activation followed by the multiplication with the weight matrix. The zeros are padded, matching the dimension when there is an increase in the dimension. We have summarized ResNet50 architecture in Table 4.1.

Layer_Type	Output_Size	Building_Blocks
conv1_*	$112X112$	$7X7$, 64, stride 2
conv2_*	$56X56$	$3X3$ max pool, stride 2
		[$1X1$, 64, $3X3$, 64, $1X1$, 256] X 3
conv3_*	$28X28$	[$1X1$, 128, $3X3$, 128, $1X1$, 512] X 4
conv4_*	$14X14$	[$1X1$, 256, $3X3$, 256, $1X1$, 1024] X 6
conv5_*	$7X7$	[$1X1$, 512, $3X3$, 512, $1X1$, 2048] X 3
-	$1X1$	average pool, 1000-d FC, softmax

Table 4.1: ResNet50 Architecture Summary [37]

MobileNetV2:

MobilNetV2 is derived from an inverted residual structure where the residual connections are between the bottleneck layers. The basic building block is bottleneck depth-separable convolutions consisting of residuals. As shown in Table 4.2, it consists of fully connected layers with 32 filters and 19 residual layers. It further employs ReLU6 as the activation function, and the kernel size is 3x3.

Input	Operator	Expansion Factor (t)	#Output Channels (c)	Repetition times (n)	Stride (s)
224^2X3	conv2d	-	32	1	2
112^2X32	bottleneck	1	16	1	1
112^2X16	bottleneck	6	24	2	2
56^2X24	bottleneck	6	32	3	2
28^2X32	bottleneck	6	64	4	2
14^2X64	bottleneck	6	96	3	1
14^2X96	bottleneck	6	160	3	2
7^2X160	bottleneck	6	320	1	1
7^2X320	conv2d $1X1$	-	1280	1	1
7^2X1280	avgpool $7X7$	-	-	1	-
$1X1X1280$	conv2d $1X1$	-	k	-	-

Table 4.2: MobileNetV2 Architecture Summary [86]

SSD_ResNet50:

The Single Shot (SS) in SSD refers to the object localization and classification tasks performed in a single forward pass of the network. The SSD network is based on a feed-forward convolutional network consisting of feature maps extraction and object detection using a convolution filter. Its uniqueness is that the final fully connected layers in the original ResNet are replaced by the SSD head, as shown in Figure 4.1. The SSD head utilizes the spatial information extracted by the ResNet to decide the bounding boxes and predict classes.

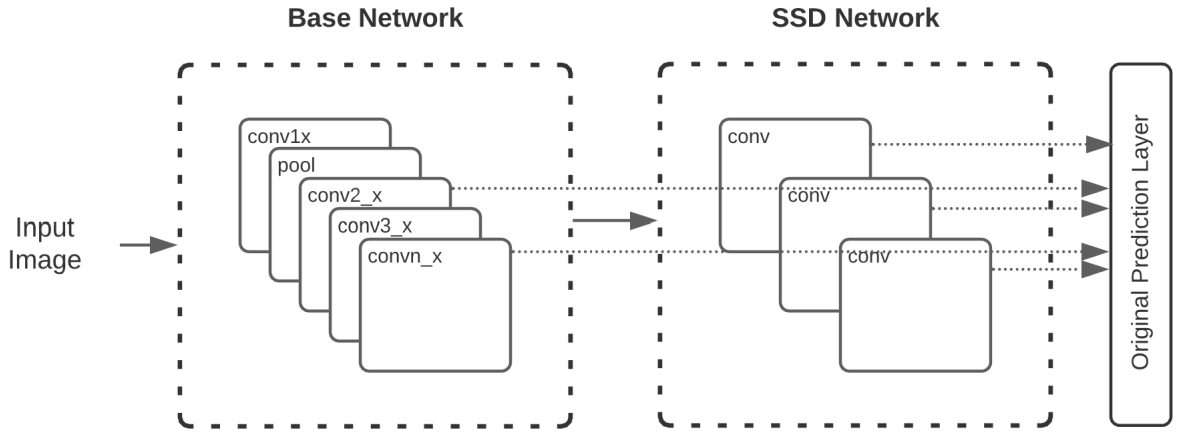


Figure 4.1: Architecture of a Convolutional Network with SSD Layers

BERT:

BERT is essentially a multi-layer bidirectional Transformer encoder based on the attention mechanism [103]. The transformer architecture employs self-attention on the encoder side and attention on the decoder end. It consists of parallel FC layers and transpose operations. Further scaling is performed before passing through the softmax layer to output probabilities. We have used $\text{BERT}_{\text{BASE}}$ extended model in this work. It has 12 layers in the encoder stack, 768 feedforward hidden units, and 12 attention heads.

Model	Train_Dataset	Test_Dataset	Framework	DL Task
ResNet50	ImageNet	ImageNet, CIFAR10	TensorFlow, PyTorch	Image Classification
MobileNetV2	ImageNet	ImageNet, CIFAR10	PyTorch	Image Classification
SSD_ResNet50	COCO	COCO	PyTorch, MXNet	Object Detection
$\text{BERT}_{\text{base_cased_squad}}$V2	SQuAD_V2	SQuAD_V2	PyTorch	Question Answering

Table 4.3: Models' Specifications

4.3.2 Compiler Optimizations Analysis

In DL compilers, the passes are categorized into optimization levels (`OPT_LEVEL`), `-Ox`, identical to the conventional compilers. We employed the domain knowledge from the neural architecture analysis to classify the optimization passes. While writing this chapter, we listed passes and their functionality available in TVM. The neural network layers, tensor operations, and their order dictated the categorization of passes. It is observed that certain passes are applicable only when a particular feature is supported by a compiler and is available in a neural network. For example, `FoldExplicitPadding` is relevant to a network with explicit padding. Using such a pass for a network like BERT will only increase the search space of the optimizations. We compiled the models under different pass combinations and examined the IR to validate this.

We studied the passes executed as part of `OPT_LEVEL=3` to establish the baseline results. Our observations supported the following two suppositions. Firstly, two or more optimization levels can produce precisely the same IR. For example, `OPT_LEVEL=2` and `OPT_LEVEL=3` generated the same IR for ResNet50. Secondly, an optimization level may contain a set of passes that do not affect the IR. For example, as part of `OPT_LEVEL=2`, `DynamicToStatic` converts dynamic operations to the static, if possible. But all the employed networks have static operations alone.

Based on the above characterization, we selected passes relevant to the experimented neural networks as summarized in Table 4 from all the available passes in TVM. We have classified them into the following broader categories:

- **Baseline Passes (BL):** These are the passes enabled as part of OPT_LEVEL=3. We have used them as our baseline experimentation.
- **ResNet Class (RN):** These passes are relevant to ResNet neural architecture.
- **MobileNet Class (MN):** These passes are explored as part of the MobileNet neural network.
- **SSD Class (SSD):** This class refers to the passes relevant to the SSD network.
- **BERT Class (BR):** These passes align with the BERT architecture.

- **Additional Passes (AD):** There are certain passes, like `ToMixedPrecision` that are dependent on the users' intent. It can be employed across networks.

This classification is intended to increase with the addition of more passes. A particular pass can belong to more than one class. While executing SSD_ResNet, we can combine RN and SSD classes to form the search space. This technique could reduce the search space for the compiler optimization selection, diminishing the overhead.

Pass	Description	Category
AlterOpLayout	Used for computing convolution in custom layouts or other general weight pre-transformation.	BL; RN; MN; SSD; BR
AnnotateSpans	Annotate a program with span information	AD
BatchingOps	Batching parallel operators into one for Conv2D, Dense and BatchMatmul	BR
CanonicalizeCast	Canonicalize cast expressions to make operator fusion more efficient.	BL; RN
CanonicalizeOps	Canonicalize special operators to basic operators	BL
CombineParallelConv2D	Combine multiple conv2d operators into one	RN
CombineParallelDense	Combine multiple dense operators into one	RN
ConvertLayout	Alternate the layouts of operators	BL
DeadCodeElimination	Remove expressions that do not have any usage	RN; MN; SSD
DefuseOps	Inverse operation of fusion pass.	BL
DynamicToStatic	Convert dynamic operations to static if possible	AD
EliminateCommonSubexpr	Eliminate common subexpressions	BL; RN; MN
FakeQuantizationToInteger	Takes fake quantized graphs and convert them to actual integer operations	AD
FastMath	Convert expensive non linear functions to their fast but approximate counterparts	MN; BR; SSD
FirstOrderGradient	Transform all global functions in the module to return the original result and the gradients of the inputs.	MN
FoldConstant	Fold the constant expressions in a Relay program	RN; MN; SSD
FoldExplicitPadding	Find explicit padding before an operator that supports implicit padding and fuses them.	RN; SSD
ForwardFoldScaleAxis	Fold the scaling of the axis into weights of conv2d/dense	BR
FuseOps	Fuse operators in an expression to a larger operator	RN; MN; SSD; BR
MergeComposite	Merge multiple operators into a single composite relay function	RN; MN; SSD
PartitionGraph	Partition a Relay program into regions that can be executed on different backends	AD
RemoveUnusedFunctions	Remove unused global relay functions in a relay module	SSD
SimplifyExpr	Simplify the Relay expression, including merging consecutive reshapes.	RN; MN; SSD
SimplifyFCTranspose	Simplify the transpose operation on a dense layer	MN; BR
SimplifyInference	Simplify the data-flow graph for the inference phase.	RN; MN; SSD
ToANormalForm	Turn Graph Normal Form expression into A Normal Form Expression	AD
ToGraphNormalForm	Turn a normal form into graph normal form.	RN; MN
ToMixedPrecision	Automatic mixed precision rewriter	AD

*BL: baseline optimization passes having OPT_LEVEL=3; RN: ResNet Class; MN: MobileNet Class; SSD: SSD_ResNet Class; BR: BERT Class; AD: Additional optimizations

Table 4.4: Categorization of Passes

4.4 Evaluation And Discussion

We executed each neural network for 100 warm-up runs and 1000 runs to gather the stats to avoid noise. Also, we considered only three standard deviations of the collected data from the mean and excluded any outliers. We selected the following metrics to assess the performance of our proposition.

- **Throughput:** the volume of inferences within a given period, usually measured in inferences per second.
- **Latency:** the execution time to perform inference on one image, expressed in milliseconds (ms).
- **Compile Time:** the time required to generate the optimized computation graph to be deployed; expressed in seconds (sec).
- **Power:** refers to the power drawn by the GPU to perform one inference. It is expressed in Watt (W),
- **Memory Used:** refers to the total memory allocated by active contexts (MiB).
- **Temperature:** refers to the core GPU temperature (°C).

The following notations are used to represent different selections and ordering of passes.

- **BL:** selection of passes having `OPT_LEVEL=3`.
- **AS-0** and **AS-1:** architecture-aware selection of passes for a given class; v0, v1. Table 4.4 is referred to get the class and varying permutations are considered to get the best result.
- **PO-0**, **PO-1**, **PO-2** and **PO-3:** randomized selection of passes; v0, v1, v2, v3. Class information and additional passes from Table 4.4 are considered to find the random set of passes and then a sequence is proposed based on multiple trials.

The pass dependency is handled internally. If a pass depends on the execution of another pass, it is called internally during the execution. In Table 4.5, we have presented the selected pass and ordering for the SSD_ResNet50 neural network in PyTorch.

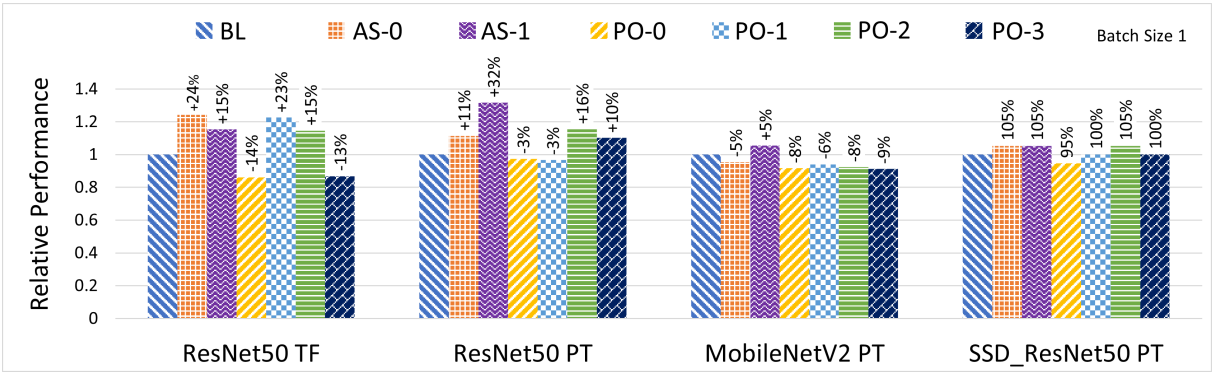
4.4.1 Experiments on GeForce RTX 2080

We evaluated the performance using seven sets of passes to formulate the pass selections similar to Table 4.5. As explained earlier, it is based on the neural network's categories presented in Table 4.4. Where AS-x is the architecture-specific selection, PO-x is the randomized selection of passes from the reduced search space. We further permuted each PO-x version to achieve the best performance among the selected version.

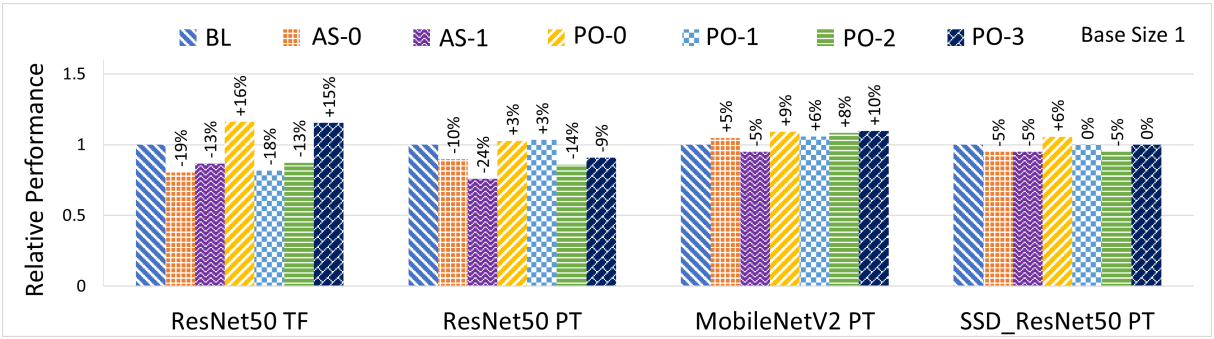
As shown in Figure 4.2a, the baseline throughput (frames/sec) for ResNet50 TF, ResNet50 PT, MobileNetV2 PT, and SSD_ResNet50 PT are 23.18, 21.35, 72.90, and 3.80, respectively. For ResNet50 implementation in TensorFlow, we achieved up to 24% improvement in the throughput with an informed selection of passes. Similar behavior was observed in the PyTorch implementation of ResNet50. Since ResNet is primarily a convolutional layers followed by the FC layer, we applied default passes like `AlterOpLayout`

ID	Selected Passes and Ordering
BL	AlterOpLayout, CanonicalizeCast, CanonicalizeOps, ConvertLayout, DefuseOps, EliminateCommonSubexpr
AS-0	AlterOpLayout, FuseOps, SimplifyExpr, FoldConstant, DeadCodeElimination, MergeComposite, FastMath, RemoveUnusedFunctions
AS-1	SimplifyExpr, FuseOps, AlterOpLayout, MergeComposite, FastMath, DeadCodeElimination, FoldConstant, RemoveUnusedFunctions
PO-0	AlterOpLayout, CombineParallelConv2D, DefuseOps, DynamicToStatic, CanonicalizeOps, CanonicalizeCast
PO-1	CanonicalizeCast, AlterOpLayout, DefuseOps, CombineParallelConv2D, PartitionGraph, FakeQuantizationToInteger
PO-2	ToMixedPrecision, CombineParallelConv2D, EliminateCommonSubexpr, SimplifyFCTranspose, CanonicalizeOps, DefuseOps, ToGraphNormalForm, ToGraphNormalForm
PO-3	CombineParallelDense, FakeQuantizationToInteger, AlterOpLayout, CombineParallelConv2D, ToGraphNormalForm, CanonicalizeOps

Table 4.5: Pass selection and ordering for SSD_ResNet50 in PyTorch



(a) Variation of "Throughput" with Pass Selection



(b) Variation of "Latency" with Pass Selection

Figure 4.2: Execution on a GeForce RTX 2080 GPU

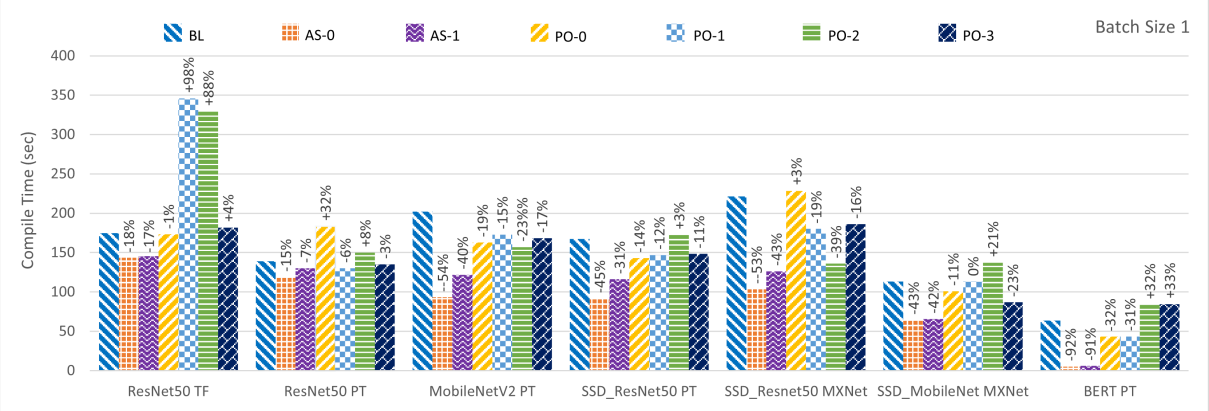


Figure 4.3: Compile-Time Reduction on A100 GPU

and `FoldConstant` followed by passes specific to the tensor operations like `FuseOps`, `EliminateCommonSubexpr`, and so on. It is observed that `FuseOps` after `AlterOpLayout` performs better as it leads to more efficient fusion. Hence, we were able to achieve an improvement of 32% in terms of throughput.

On the contrary, throughput for MobileNet and SSD-ResNet did not improve much on a non-tensor core architecture. There was a 5% improvement on average. MobileNet is a lightweight architecture primarily consists of bottleneck layers containing fewer nodes than the previous layer. Hence, this network class does not have many relevant passes. Similarly, for the SSD_ResNet, the addition of new SSD layers reduces the overall gain. Analogous behavior was noticed with the latency gain. As shown in Figure 4.2b, the baseline latency (ms/inference) for ResNet50 TF, ResNet50 PT, MobileNetV2 PT, and SSD_ResNet50 PT are 43.13, 46.83, 13.71, and 263.16, respectively. The neural architecture-aware selection of optimizations reduced the latency by 18%-19% in the case of ResNet50 in TF, and up to 24% in PyTorch implementation, as shown in Figure 4.2b. We achieved up to a 5% latency gain for the MobileNet and SSD_ResNet models.

4.4.2 Experiments on A100

The NVIDIA A100 GPU is a tensor core supported hardware that offers advanced support for tensor operations, mainly CUDA graphs acceleration, as discussed before. As shown in Figure 4.3, pruning the optimization passes search space and selecting lesser and more relevant passes than `-O3 OPT_LEVEL` improves the compilation time by 18% in ResNet50 TF and by 15% in the PyTorch implementation. Unlike PyTorch, on using combinations of `MergeCompilerRegions` and `SimplifyInference`, the compile-time in TF almost doubled the baseline compile-time. It is due to the aggressive traversal of the computation graph without optimizing the IR in TF. Currently, we are investigating the detailed cause of this behavior. One possible explanation is the difference in the implementation of operators in TF and PyTorch, and hence the difference in the computation graph generated in either case.

The architecture-aware selection of passes was paramount to MobileNet and SSD_ResNet. In both scenarios, we could reduce compile-time by almost 45%-54%. As shown in Table 4.4 and 4.5, the fewer optimization passes reduced the overhead of traversing a vast computation graph. While we could cherry-pick passes like `MergeComposite`, and

`FastMath`, passes like `DynamicToStatic` and `FakeQuantizationToInteger` could be excluded from the AS-x versions. We validated our results across different frameworks. SSD_MobileNet implementation in MXNet showed similar behavior. On evaluating different classes of models, we found that the new tensor core offers hardware optimizations that diminish the benefits performed by various compiler optimization passes selected as part of `-O3 OPT_LEVEL`. Since A100 comes with tensor cores and advanced CUDA compute capabilities as described in section 4.1, we also evaluated the computation graph without any optimization passes in scenarios where the hardware would do most optimizations. For SSD-based models across formats, it performed on par with baseline execution.

The most exciting results came from the BERT's architecture-aware selection of passes. We could achieve an almost 92% reduction in the compile-time without reducing the throughput. That is critical when we need to Just-in-Time (JIT) compile for the edge devices. We found that only a few passes were relevant due to the BERT-based model's transformer-based architecture. Hence we narrowed down the search space to the BR class passes ad selected passes like `SimplifyFCTranspose`, `FastMath`, `FoldExplicitPadding`. BERT employs three parallel FC layers followed by three parallel transpose operations in a self-attention layer. Also, it performs scaling and softmax that get benefited by including `FastMath`. Furthermore, it uses padding extensively for the shorted inputs.

Compute Hardware	Model	Framework	Pass Order	Power (W)		Temperature (°C)		Memory Util. (MiB)	
				Median	Max	Median	Max	Median	Max
GeForce RTX 2080	ResNet50_V2	TensorFlow	`-O3`	48	50	29	32	601	650
			AS $_{best}$	**43**	**46**	28	32	**592**	**638**
		PyTorch	`-O3`	47	49	33	35	456	504
			AS $_{best}$	46	49	**29**	**32**	**433**	504
	SSD_ResNet50	PyTorch	`-O3`	90	134	31	37	1287	1458
			AS $_{best}$	**83**	133	**30**	**34**	**1160**	1450
	MobileNet_V2	PyTorch	`-O3`	42	45	28	31	174	190
			AS $_{best}$	**41**	**43**	28	29	**154**	**186**
A100	SSD_ResNet50	MXNet	`-O3`	80	119	26	30	1747	1887
			AS $_{best}$	**73**	**110**	**25**	**27**	**1723**	**1879**
	MobileNet_V2	PyTorch	`-O3`	59	63	23	26	1552	1650
			AS $_{best}$	58	63	23	25	**1540**	1650
	BERT	PyTorch	`-O3`	109	249	26	36	1782	2586
			AS $_{best}$	104	250	26	36	**1680**	2586

Table 4.6: Selected Models with Considerable Distinctions

*AS $_{best}$: Neural Architecture-Aware selection of passes.

Additionally, we gathered hardware statistics. Namely, power consumption, GPU temperature, and memory utilization, to quantify the architecture-aware pass selection and its effect on the earlier metrics. The selected results are summarized in Table 4.6. On GeForce, where the peak memory utilization remained almost identical for all the experiments, a 10% decrease in the median memory utilization is reported in the SSD_ResNet and a 5% in ResNet50 PyTorch implementation. Power consumption exhibited equivalent behavior. On A100, memory utilization was reduced by 2%-6% across all the runs. The observations confirm that the architecture-aware selection of passes impacts memory utilization and power consumption, an essential aspect of computation on resource-constrained edge devices.

4.5 Related Work

Pass selection and phase-ordering problems for the compiler writers are decades old but pertinent. With the development of graph-based deep learning compilers, there are manifold possibilities for multilayered optimizations targeting computation graphs generated from an input framework like TensorFlow or PyTorch. After optimizations are applied, the resultant intermediate representation (IR) differs significantly, affecting the overall performance. Since optimization passes depend on various factors, including the code block, backend architecture characteristics, and the compiler itself, the search space is enormous, making the selection of passes and ordering an *NP-hard* problem.

In work performed by Haneda et al. [36], the authors propose a statistical technique to reduce the search space for the compiler passes. They evaluated SPECint95 benchmark suite [19] execution on a GCC compiler, validating the heuristics. Similarly, Kulkarni et al. [50] suggest a careful and aggressive search space pruning without any information loss. It analyzes the probabilities of various phase interactions, such as inter-phase enabling/disabling relationships and inter-phase independence. Furthermore, research groups [45] have also investigated methodologies involving manually partitioning the optimization phases into independent groups to develop a new multi-stage search algorithm. On average, the performed iterative technique could achieve an 89% reduced search space.

Besides the statistical and iterative heuristics, researchers have also designed machine learning-based heuristics [5, 47] to get to the bottom of efficient execution order. In another work[6], the authors employ clustering-based predictive modeling using dynamic features to attack the problem. They evaluate the results on the Ctuning CBench suite [31] against other earlier discussed heuristics methods. Additionally, a research group implemented a reinforcement learning (RL) based LLVM-derived framework, Autophase [39], to deal with the phase-ordering problem for High-Level Synthesis (HLS) programs.

The effect of phase ordering on energy and power consumption is also investigated for the LLVM-based compilers [73, 32]. The experiments exhibit a weak correlation between energy consumption and performance, albeit the authors could significantly decrease the energy consumption and execution time in specific scenarios.

Almost every work discussed so far targets LLVM-based compilers. Currently, DL compilers employ predefined flags, `-O2`, `-O3`, etc. In this work, we show that the search space can be pruned at a higher level, reducing efforts to auto-tune if we make a neural-architecture aware selection of passes resulting in reduced execution time and improved throughput. Additionally, it is more relevant to have a compiler-agnostic heuristic involving domain knowledge from neural architecture instead of conventional `-Ox` optimization levels.

Chapter 5

Conclusion and Future Work

In this report, we have discussed the tensor compilers for inference, and various optimizations offered by them and have performed a comparative analysis. There is a clear need to optimize DL models on edge devices for better latency and power performance. TensorFlow Lite (TFLite) and TensorFlow-TenorRT (TF-TRT) are considered state-of-the-art inference compilers for edge computing. This chapter presents a detailed performance study of TFLite and TF-TRT using commonly employed DL models for edge devices on varying hardware platforms. We find that TF-TRT integration performs better at high precision mode but loses its edge for model compression to TFLite at low precision mode. TF-TRT consistently performs better with different DL architectures, especially with GPUs using tensor cores. However, TFLite performs better with lightweight DL models than with deep neural network-based models.

We have further proposed in Chapter 3 HPC ontology and a framework, HPCFair, to modularize the design space exploration given various data objects for apple-to-apple comparison and speed up the tensor compiler research. With the proliferating AI research and development among the HPC and scientific community, the need for a platform to contain data objects in an easily findable, accessible format, enabling interconvertibility and reusability among various frameworks, is indispensable. HPCFair is an attempt in that direction. We have endeavored to abridge gaps in different state-of-the-art frameworks. We exhibited how HPCFair can assist development from a baseline DNN experiment to individual ML Libraries as reusable components, extending to reproduce research results and pipeline development. We also demonstrated how it could be leveraged to expedite and ease design-space exploration. HPCFair is our first step in applying FAIR principles to HPC applications. Where we support only TF-ONNX interconversion and PyTorch-ONNX conversion, the eventual intention is to implement checkpointing conversion-like methodology. It will equip the user with the ability to make TF-PyTorch models interoperable.

Lastly, in Chapter 4, we have presented how neural architecture aware classification and selection of passes can improve the compilation time by significant orders. We exhibited how the proposed approach can prune the search space and significantly reduce compile time. It can be substantial for applications heavily dependent on JIT compilation, like edge computing. Also, with an increasing number of passes and the complexity of the computation graph, it is essential to reduce search space to facilitate static rule-based or ML technique-based pass selection. In the future, we plan to propose an intelligent methodology from the computation graph and optimize the selection procedure on a

resource-constrained edge device, emphasizing the metrics like power consumption and device temperature.

5.1 Future Work

We have demonstrated how optimizations offered by the tensor compilers can improve the overall performance of the deep learning workload in a server-edge hybrid environment. At the same time, keeping up with the emerging ML/DL and hardware is a challenge. It motivates us further to improve the compilation with these possible future steps:

- Experimentation with scientific workload: The scientific computing community is considering edge computing for its ML workload to analyze real-time data during experiments. The performance of inference compilers such as TensorRT and TensorFlow Lite on varying hardware (ASICs, TPU, etc.) under a scientific ML workload is an essential question for the scientific community.
- Usage of hardware metrics: Use metrics in addition to the existing ones, like performance per watt (or custom metric), to better account for energy efficiency.
- We are improving benchmarking and interoperability of compiler optimizations across diverse and continually emerging software and hardware from servers to embedded devices.
- Automate tensor program generation: Auto-tuning the search-based tensor compilers for deep learning workloads is gaining attention. It has resulted in machine learning-derived cost modeling advancements to automate tensor program generation. The immense search space makes it arduous to have specific data-intensive cost models for individual hardware. Hence, a strategy that can efficiently generate tensor programs on a heterogeneous device is desired.

Bibliography

[1] Martın Abadi 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.

[2] *Akraino*. 2018. URL: https://www.lfedge.org/projects/akraino/.

[3] *Apache Edgent*. URL: https://edgent.incubator.apache.org/.

[4] ARM. *ARM Neon Architecture*. URL: https://developer.arm.com/documentation/dht0002/a/Introducing-NEON/NEON-architecture-overview/NEON-instructions.

[5] Amir Hossein Ashouri et al. "Predictive modeling methodology for compiler phase-ordering". In: *Proceedings of the 7th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and the 5th Workshop on Design Tools and Architectures For Multicore Embedded Computing Platforms*. 2016, pp. 7–12.

[6] Amir H Ashouri et al. "Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 14.3 (2017), pp. 1–28.

[7] *Azure IoT Edge*. URL: https://github.com/Azure/iotedge.

[8] Rafael Ballester-Ripoll, Enrique G Paredes, and Renato Pajarola. "Sobol tensor trains for global sensitivity analysis". In: *Reliability Engineering & System Safety* 183 (2019), pp. 311–322.

[9] Ketan Bhardwaj et al. "Fast, scalable and secure onloading of edge functions using airbox". In: *2016 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE. 2016, pp. 14–27.

[10] Andrea Borghesi et al. "Anomaly detection using autoencoders in high performance computing systems". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 2019, pp. 9428–9433.

[11] Cristian Bucilua, Rich Caruana, and Alexandru Niculescu-Mizil. "Model compression". In: *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2006, pp. 535–541.

[12] *Caffe Model Zoo*. URL: https://caffe.berkeleyvision.org/model_zoo.html.

[13] R. Chard et al. "DLHub: Model and Data Serving for Science". In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2019, pp. 283–292. DOI: 10.1109/IPDPS.2019.00038. URL: https://doi.ieeecomputersociety.org/10.1109/IPDPS.2019.00038.

[14] Tianqi Chen et al. "{TVM}: An automated end-to-end optimizing compiler for deep learning". In: *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 2018, pp. 578–594.

[15] Tianqi Chen et al. "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems". In: *arXiv preprint arXiv:1512.01274* (2015).

[16] *CLgen*. June 2021. URL: https://chriscummins.cc/clgen/.

[17] *Collective Knowledge Framework*. June 2021. URL: http://cknowledge.org/.

[18] *Cord*. 2018. URL: https://www.opennetworking.org/cord.

[19] Standard Performance Evaluation Corporation. *SPECint95 Benchmark Suite*. Feb. 1995-2022. URL: https://www.spec.org/cpu95/CINT95/index.html.

[20] Chris Cummins et al. "End-to-end deep learning of optimization heuristics". In: *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE. 2017, pp. 219–232.

[21] Scott Cyphers et al. "Intel ngraph: An intermediate representation, compiler, and executor for deep learning". In: *arXiv preprint arXiv:1801.08058* (2018).

[22] Anwesha Das et al. "Doomsday: predicting which node will fail when on supercomputers". In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2018, pp. 108–121.

[23] Michael V DeBole et al. "TrueNorth: Accelerating from zero to 64 million neurons in 10 years". In: *Computer* 52.5 (2019), pp. 20–29.

[24] Emily L Denton et al. "Exploiting linear structure within convolutional networks for efficient evaluation". In: *Advances in neural information processing systems* 27 (2014), pp. 1269–1277.

[25] *Department of Energy Announces 8.5 Million for FAIR Data to Advance Artificial Intelligence for Science*. URL: https://www.energy.gov/articles/department-energy-announces-85-million-fair-data-advance-artificial-intelligence-science.

[26] Jacob Devlin et al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[27] Unai Elordi et al. "Benchmarking Deep Neural Network Inference Performance on Serverless Environments With MLPerf". In: *IEEE Softw.* 38.1 (2021), pp. 81–87. DOI: 10.1109/MS.2020.3030199. URL: https://doi.org/10.1109/MS.2020.3030199.

[28] Murali Emani et al. "Accelerating scientific applications with sambanova reconfigurable dataflow architecture". In: *Computing in Science & Engineering* 23.2 (2021), pp. 114–119.

[29] *Facebook Glow*. URL: https://ai.facebook.com/tools/glow/.

[30] Kjiersten Fagnan et al. *Data and models: a framework for advancing AI in science*. Tech. rep. USDOE Office of Science (SC)(United States), 2019.

[31] Grigori Fursin and Olivier Temam. "Collective optimization: A practical collaborative approach". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 7.4 (2010), pp. 1–29.

[32] Kyriakos Georgiou et al. "Less is more: Exploiting the standard compiler optimization levels for better performance and energy consumption". In: *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*. 2018, pp. 35–42.

[33] *GluonCV Model Zoo*. URL: https://cv.gluon.ai/model_zoo/index.htmlx.

[34] Google. *TensorFlow Lite*. URL: https://www.tensorflow.org/lite.

[35] Dominik Grewe, Zheng Wang, and Michael FP O'Boyle. "Portable mapping of data parallel programs to opencl for heterogeneous systems". In: *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2013, pp. 1–10.

[36] Masayo Haneda, Peter MW Knijnenburg, and Harry AG Wijshoff. "Optimizing general purpose compiler optimization". In: *Proceedings of the 2nd conference on Computing frontiers*. 2005, pp. 180–188.

[37] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[38] Chuang Hu et al. "Dynamic adaptive DNN surgery for inference acceleration on the edge". In: *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE. 2019, pp. 1423–1431.

[39] Qijing Huang et al. "Autophase: Compiler phase-ordering for hls with deep reinforcement learning". In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2019, pp. 308–308.

[40] Yuanjie Huang et al. "Transforming GCC into a research-friendly environment: plugins for optimization tuning and reordering, function cloning and program instrumentation". In: *2nd International Workshop on GCC Research Opportunities (GROW'10)*. 2010.

[41] *Hugging Face*. June 2021. URL: https://huggingface.co/.

[42] Loc N Huynh, Rajesh Krishna Balan, and Youngki Lee. "Deepmon: Building mobile gpu deep learning models for continuous vision applications". In: *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. 2017, pp. 186–186.

[43] *ImageNet*. URL: http://image-net.org/index.

[44] Tanzima Islam et al. "Toward a Programmable Analysis and Visualization Framework for Interactive Performance Analytics". In: *2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*. IEEE. 2019, pp. 70–77.

[45] Michael R Jantz and Prasad A Kulkarni. "Exploiting phase inter-dependencies for faster iterative compiler optimization phase order searches". In: *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. IEEE. 2013, pp. 1–10.

[46] Sanath Jayasena et al. "Detection of false sharing using machine learning". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2013, pp. 1–9.

[47] Tarindu Jayatilaka et al. "Towards Compile-Time-Reducing Compiler Optimization Selection via Machine Learning". In: *50th International Conference on Parallel Processing Workshop*. 2021, pp. 1–6.

[48] Zhe Jia et al. "Dissecting the graphcore ipu architecture via microbenchmarking". In: *arXiv preprint arXiv:1912.03413* (2019).

[49] Norman P Jouppi et al. "In-datacenter performance analysis of a tensor processing unit". In: *Proceedings of the 44th annual international symposium on computer architecture*. 2017, pp. 1–12.

[50] Prasad A Kulkarni et al. "Practical exhaustive optimization phase order exploration and evaluation". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 6.1 (2009), pp. 1–36.

[51] Chris Lattner et al. "MLIR: A Compiler Infrastructure for the End of Moore's Law". In: 2020. arXiv: 2002.11054 [cs.PL].

[52] En Li, Zhi Zhou, and Xu Chen. "Edge intelligence: On-demand deep learning model co-inference with device-edge synergy". In: *Proceedings of the 2018 Workshop on Mobile Edge Communications*. 2018, pp. 31–36.

[53] Mingzhen Li et al. "The Deep Learning Compiler: A Comprehensive Survey". In: *arXiv preprint arXiv:2002.03794* (2020).

[54] Chunhua Liao et al. "DataRaceBench: a benchmark suite for systematic evaluation of data race detection tools". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2017, pp. 1–14.

[55] Chunhua Liao et al. "HPC Ontology: Towards a Unified Ontology for Managing Training Datasets and AI Models for High-Performance Computing". In: *2021 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC)*. IEEE. 2021, pp. 69–80.

[56] Tsung-Yi Lin et al. "Microsoft coco: Common objects in context". In: *European conference on computer vision*. Springer. 2014, pp. 740–755.

[57] Fang Liu et al. "A survey on edge computing systems and tools". In: *Proceedings of the IEEE* 107.8 (2019), pp. 1537–1562.

[58] Wei Liu et al. "SSD: Single shot multibox detector". In: *European conference on computer vision*. Springer. 2016, pp. 21–37.

[59] Raphael Gontijo Lopes, Stefano Fenu, and Thad Starner. "Data-free knowledge distillation for deep neural networks". In: *arXiv preprint arXiv:1710.07535* (2017).

[60] Unai Lopez-Novoa, Alexander Mendiburu, and Jose Miguel-Alonso. "A survey of performance modeling and simulation techniques for accelerator-based computing". In: *IEEE Transactions on Parallel and Distributed Systems* 26.1 (2014), pp. 272–281.

[61] Alberto Magni, Christophe Dubach, and Michael O'Boyle. "Automatic optimization of thread-coarsening for graphics processors". In: *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 2014, pp. 455–466.

[62] Preeti Malakar et al. "Benchmarking machine learning methods for performance modeling of scientific applications". In: *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE. 2018, pp. 33–44.

[63] Alberto Marchisio et al. "Deep learning for edge computing: Current trends, cross-layer optimizations, and open research challenges". In: *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE. 2019, pp. 553–559.

[64] Luiz GA Martins et al. "A clustering-based approach for exploring sequences of compiler optimizations". In: *2014 IEEE Congress on Evolutionary Computation (CEC)*. IEEE. 2014, pp. 2436–2443.

[65] Luiz GA Martins et al. "Clustering-based selection for the exploration of compiler optimization sequences". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 13.1 (2016), pp. 1–28.

[66] Amrita Mathuriya et al. "CosmoFlow: Using deep learning to learn the universe at scale". In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2018, pp. 819–829.

[67] *MLCube*. June 2021. URL: `https://mlcommons.org/en/mlcube/`.

[68] *MNIST Dataset*. June 2021. URL: `http://yann.lecun.com/exdb/mnist/`.

[69] Boris Motik et al. "Owl 2 web ontology language: Structural specification and functional-style syntax". In: *W3C recommendation* 27.65 (2009), p. 159.

[70] Mena Nagiub and Wael Farag. "Automatic selection of compiler options using genetic techniques for embedded software design". In: *2013 IEEE 14th International Symposium on Computational Intelligence and Informatics (CINTI)*. 2013, pp. 69–74. DOI: `10.1109/CINTI.2013.6705166`.

[71] Ricardo Nobre, Luiz GA Martins, and Joao MP Cardoso. "Use of previously acquired positioning of optimizations for phase ordering exploration". In: *Proceedings of the 18th international workshop on software and compilers for embedded systems*. 2015, pp. 58–67.

[72] Ricardo Nobre, Luiz GA Martins, and João MP Cardoso. "A graph-based iterative compiler pass selection and phase ordering approach". In: *ACM SIGPLAN Notices* 51.5 (2016), pp. 21–30.

[73] Ricardo Nobre, Luıs Reis, and Joao MP Cardoso. "Compiler phase ordering as an orthogonal approach for reducing energy consumption". In: *arXiv preprint arXiv:1807.00638* (2018).

[74] Cedric Nugteren and Valeriu Codreanu. "CLTune: A generic auto-tuner for OpenCL kernels". In: *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*. IEEE. 2015, pp. 195–202.

[75] NVIDIA. *DeepStream SDK*. URL: `https://developer.nvidia.com/deepstream-sdk`.

[76] NVIDIA. *TensorRT*. v8.0.1. URL: `https://developer.nvidia.com/tensorrt`.

[77] *ONNX*. June 2021. URL: `https://onnx.ai/`.

[78] *ONNX Model Zoo*. URL: `https://github.com/onnx/models`.

[79] Adam Paszke et al. "Pytorch: An imperative style, high-performance deep learning library". In: *Advances in neural information processing systems* 32 (2019).

[80] Caleb Phillipsy. *HPC Energy Research*. 2016. URL: `https://cscdata.nrel.gov/#/datasets/d332818f-ef57-4189-ba1d-beea291886eb%22`.

[81] Riccardo Poggi. "Digital Signal Processing in FPGA for Particle Track Reconstruction at the HL-LHC ATLAS". In: *8th International Conference on Modern Circuits and Systems Technologies, MOCAST 2019, Thessaloniki, Greece, May 13-15, 2019*. IEEE, 2019, pp. 1–4. DOI: 10.1109/MOCAST.2019.8741949. URL: https://doi.org/10.1109/MOCAST.2019.8741949.

[82] *PyTorch Hub*. June 2021. URL: https://pytorch.org/hub/.

[83] Nadav Rotem et al. "Glow: Graph lowering compiler techniques for neural networks". In: *arXiv preprint arXiv:1805.00907* (2018).

[84] Amit Sabne. *XLA : Compiling Machine Learning for Peak Performance*. 2020.

[85] Hooman Peiro Sajjad et al. "Spanedge: Towards unifying stream processing over central and near-the-edge data centers". In: *2016 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE. 2016, pp. 168–178.

[86] Mark Sandler et al. "Mobilenetv2: Inverted residuals and linear bottlenecks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 4510–4520.

[87] Mahadev Satyanarayanan et al. "The case for vm-based cloudlets in mobile computing". In: *IEEE pervasive Computing* 8.4 (2009), pp. 14–23.

[88] *SGEMM GPU kernel performance Data Set*. June 2021. URL: https://archive.ics.uci.edu/ml/datasets/SGEMM+GPU+kernel+performance.

[89] Rupal Shrivastava. *GPU Kernel Performance Dataset*. https://www.kaggle.com. 2020.

[90] Karan Singh et al. "Predicting parallel application performance via machine learning approaches". In: *Concurrency and Computation: Practice and Experience* 19.17 (2007), pp. 2219–2235.

[91] *SPARQL for Tables: Turn CSV into RDF using SPARQL syntax*. https://tarql.github.io/. Accessed: 2021-06-09.

[92] Manu Sporny et al. "JSON-LD 1.0". In: *W3C recommendation* 16 (2014), p. 41.

[93] Nitin Sukhija et al. "A learning-based selection for portfolio scheduling of scientific applications on heterogeneous computing systems". In: *Parallel and Cloud Computing* 3.4 (2014), pp. 66–81.

[94] Jingwei Sun et al. "Automated performance modeling based on runtime feature detection and machine learning". In: *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*. IEEE. 2017, pp. 744–751.

[95] *Tensorflow Hub*. June 2021. URL: https://www.tensorflow.org/hub.

[96] *Tensorflow XLA*. URL: https://www.tensorflow.org/xla.

[97] Jayaraman J Thiagarajan et al. "PADDLE: Performance Analysis Using a Data-Driven Learning Environment". In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2018, pp. 784–793.

[98] *Top500 List, June 2021*. https://www.top500.org/lists/top500/2021/06/. June, 2021.

[99] Ozan Tuncer et al. "Diagnosing performance variations in HPC applications using machine learning". In: *International Supercomputing Conference*. Springer. 2017, pp. 355–373.

[100] *Uno*. URL: https://github.com/ECP-CANDLE/Benchmarks/tree/develop/Pilot1/Uno.

[101] Mike Uschold and Michael Gruninger. "Ontologies: Principles, methods and applications". In: *The knowledge engineering review* 11.2 (1996), pp. 93–136.

[102] Han Vanholder. *Efficient Inference with TensorRT*.

[103] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[104] Gaurav Verma et al. "HPCFAIR: Enabling FAIR AI for HPC Applications". In: *2021 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC)*. IEEE. 2021, pp. 58–68.

[105] Gaurav Verma et al. "Performance evaluation of deep learning compilers for edge inference". In: *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2021, pp. 858–865.

[106] Gaurav Verma et al. "Towards neural architecture-aware exploration of compiler optimizations in a deep learning {graph} compiler". In: *Proceedings of the 19th ACM International Conference on Computing Frontiers*. 2022, pp. 244–250.

[107] Jeffrey Vetter. "Performance analysis of distributed applications using automatic classification of communication inefficiencies". In: *Proceedings of the 14th international conference on Supercomputing*. 2000, pp. 245–254.

[108] Xiaying Wang et al. "Fann-on-mcu: An open-source toolkit for energy-efficient neural network inference at the edge of the internet of things". In: *IEEE Internet of Things Journal* 7.5 (2020), pp. 4403–4417.

[109] Mark D Wilkinson et al. "The FAIR Guiding Principles for scientific data management and stewardship". In: *Scientific data* 3.1 (2016), pp. 1–9.

[110] Diana Wofk et al. "Fastdepth: Fast monocular depth estimation on embedded systems". In: *2019 International Conference on Robotics and Automation (ICRA)*. IEEE. 2019, pp. 6101–6108.

[111] Carole-Jean Wu et al. "Machine learning at facebook: Understanding inference at the edge". In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2019, pp. 331–344.

[112] Hailu Xu et al. "Machine learning guided optimal use of GPU unified memory". In: *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. IEEE. 2019, pp. 64–70.

[113] Matei Zaharia et al. “Accelerating the machine learning lifecycle with MLflow.” In: *IEEE Data Eng. Bull.* 41.4 (2018), pp. 39–45.

[114] Yue Zhao et al. “Bridging the gap between deep learning and sparse matrix format selection”. In: *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*. 2018, pp. 94–108.

[115] Barret Zoph et al. “Learning transferable architectures for scalable image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 8697–8710.