ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

SDSC

Data Storage and Ingestion for Single Node Deep Learning

Bachelor's Thesis

Jan Hochstrasser
jhochstrasse@ethz.ch

August 28, 2024

Advisors: Prof. Dr. Fernando Perez-Cruz
Co-Supervisors: Dr. Firat Ozdemir, Dr. Matthias Meyer
Swiss Data Science Center (SDSC), ETH Zurich and EPFL

Abstract

Deep learning (DL) requires large amounts of data to reach state-of-the-art performance [1], necessitating a high data throughput to avoid inefficient and costly stalls of the computing infrastructure. High-performance data storage and ingestion (DSI) methods are crucial for this but frequently overlooked.

Current literature on DSI performance is often limited to artificial data [2] or large-scale data [3] that is out of reach for medium-sized companies and most academic groups. Further, studies commonly examine data loaders [4] or storage formats [2] in isolation. This is insufficient, as our results show. The two joint components impact DSI performance for a specific data set.

To tackle the shortcomings above, we developed the open-source framework DSI-Bench [5] to measure DSI performance on a single-node-scale using multiple real-world data sets [6–9]. They are converted into various storage formats [10–14] and ingested by consumers employing data loaders from different DL frameworks [15–17]. Comprehensive statistics are generated for execution time, CPU load, RAM utilization, and required disk space. DSI-Bench is also easily extensible for custom use cases, enabling a well-founded choice of the best DSI option given a specific situation.

The results generated by DSI-Bench show the importance of doing this type of benchmark. The combination of storage format, data loader, and data set determines the throughput, CPU load, RAM utilization, and required disk space. The performance can vary substantially, so choosing a better option saves resources and costs by reducing training time and stalls in computing infrastructure.

Acknowledgments

First and foremost, I would like to thank the co-supervisors of my Bachelor's Thesis, Dr. Firat Ozdemir and Dr. Matthias Meyer. They were always open to rapidly helping me with any difficulties during the project. The various insights and ideas I received from them throughout my thesis proved to be very valuable. I am grateful that I was able to profit from their experience.

I would also like to thank my friends and family, especially Elena Koller and Sophia Herrmann. They were often forced to listen to me talk about my thesis, and although they usually couldn't help me directly, their listening helped me immensely and was more than I could have asked for.

Contents

List of Figures

List of Tables

List of Configuration Files

Abbreviations

DL deep learning

DSI data storage and ingestion

PyN Python native

TF TensorFlow

sTF sequential TensorFlow

PyT PyTorch

DPyT DALI PyTorch

DTF DALI TensorFlow

TFRs TFrecords single

TFRm TFRecords multi

CLI command line interface

i.i.d. independent and identically distributed

Chapter 1

Introduction

1.1 Introduction

With the progression of deep learning (DL), we saw that larger data sets enable us to achieve better results, for example in vision tasks [1]. Over time, a growing amount of compute was used to train different models [18] with more data. With the need for these large quantities of computational power and GPUs being a major bottleneck for training duration over a long time, a lot of work and effort went into increasing their speed and efficiency. A result of these efforts can be seen when comparing the NVIDIA H100 tensor core GPU architecture to the previous A100 tensor core GPU architecture. A speedup between 2 and 9 can be observed for artificial intelligence training [19].

The focus on computational power consequently led to data storage and ingestion (DSI) being overlooked and growing into a major bottleneck [20]. If training infrastructure can process data faster than it receives it, the devices start to stall while waiting for the data. This can happen due to expensive CPU-based preprocessing, limited storage reading speed, or other reasons. Stalls are inefficient and costly as compute resources are wasted. With the spread of DL applications in several domain sciences [21], the urgency of the described issue further intensifies.

Using efficient DSI methods is increasingly important to prevent data stalls and reduce costs and training time. Using a comprehensive benchmark, data scientists could choose the best-suited option for their needs and make their training process more efficient. For this reason, we introduce DSI-Bench, a benchmarking framework to compare the performance of different DSI implementations using real-world data sets.

1.2 Related Work

There are different papers on DSI performance (e.g. [2–4, 22]), but the literature landscape is limited. The studies are commonly restricted to small, artificial data sets [2] or data loaders and storage formats in isolation, disregarding their symbiosis [2, 4]. Recently, there has been a comprehensive study on DSI systems [3], but it focuses on large-scale DL models. These scales are out of reach for small to medium-sized companies and most academic research groups.

There have also been studies on a smaller to medium scale [2, 23], but these typically concentrate on one part of the DSI pipeline. This misses the opportunity to compare combinations of different components and how they influence each other's performance.

Several benchmarking frameworks already exist in the space of data science and machine learning [4, 22]. These are extensive and very useful but usually focus on the performance of different algorithms [22] or the evaluation of a single part of the DSI pipeline [4], as opposed to the combination of all included components and their reciprocal influence.

Based on the previous observations, we conclude the necessity of performing benchmarks on real-world data sets, focusing on the interaction of data loaders and storage formats. Such a benchmark designed for single node scales allows various groups of the data science community to choose the best-suited option for their specific situation.

1.3 Our Contribution

During this thesis, we developed the comprehensive benchmarking framework DSI-Bench [5]. The main objectives of the framework are to provide a possibility to obtain realistic benchmark results for different DSI options, be easy to use, and adapt to individual use cases. With these goals in mind, we want to give data scientists the opportunity to choose the most efficient combination of data loader and storage format for their projects.

To achieve these goals, DSI-Bench allows the performance comparison of different storage formats and data loaders for various data sets. Using real-world data sets, unlike artificial data, allows for a more realistic benchmark result. To accommodate multiple constraining factors, runtime, CPU load and RAM utilization are measured during the data set consumption. The size used on disk by the different data set conversions is also tracked.

The structure of the framework (Section 2.2) allows data scientists to easily integrate their own data sets, consumers, and storage formats into the benchmark (Section 2.4). The possibility for extension makes DSI-Bench flexible. The possibility of using a project's data set ensures the most accurate result for a concrete scenario. As shown by the results, the best combination of data loader and storage format does vary from data set to data set, signifying the importance of using the actual data set.

The available implementations of data sets (Section 2.3.3), consumers (Section 2.3.2), and storage formats (Section 2.3.1) are illustrated in Tables 1.1 to 1.5. The following abbreviations are used in the tables:

- Python native (PyN): the Python native consumer
- TensorFlow (TF): the TensorFlow consumer
- sequential TensorFlow (sTF): the sequential TensorFlow consumer
- PyTorch (PyT): the PyTorch consumer

- DALI PyTorch (DPyT): the NVIDIA DALI PyTorch consumer
- DALI TensorFlow (DTF): the NVIDIA DALI TensorFlow consumer
- TFrecords single (TFRs): the TFRecords single storage format
- TFRecords multi (TFRm): the TFRecords multi storage format

Table 1.1: Compatibility overview for TEST data set

TEST

	PyN	TF	sTF	PyT	DPyT	DTF
Raw (in memory)	✓	✓	✓	✓	✗	✗
HDF5	✓	✓	✓	✓	✗	✗
Zarr	✓	✓	✓	✓	✗	✗
Pickle	✓	✓	✓	✓	✗	✗
NPY	✓	✓	✓	✓	✓	✓
NPZ	✓	✓	✓	✓	✗	✗
TFRs	✓	✓	✓	✓	✓	✓
TFRm	✓	✓	✓	✓	✗	✗

Table 1.2: Compatibility overview for OADAT data set

OADAT

	PyN	TF	sTF	PyT	DPyT	DTF
Raw (HDF5)	✓	✓	✓	✓	✗	✗
HDF5	✓	✓	✓	✓	✗	✗
Zarr	✓	✓	✓	✓	✗	✗
Pickle	✓	✓	✓	✓	✗	✗
NPY	✓	✓	✓	✓	✓	✓
NPZ	✓	✓	✓	✓	✗	✗
TFRs	✓	✓	✓	✓	✓	✓
TFRm	✓	✓	✓	✓	✗	✗

Table 1.3: Compatibility overview for ICON data set

ICON

	PyN	TF	sTF	PyT	DPyT	DTF
Raw (HDF5)	✓	✓	✓	✓	✗	✗
HDF5	✓	✓	✓	✓	✗	✗
Zarr	✓	✓	✓	✓	✗	✗
Pickle	✓	✓	✓	✓	✗	✗
NPY	✓	✓	✓	✓	✓	✓
NPZ	✓	✓	✓	✓	✗	✗
TFRs	✓	✗	✗	✓	✓	✓
TFRm	✓	✗	✗	✓	✗	✗

Table 1.4: Compatibility overview for ILSVRC 2012 data set

ILSVRC 2012						
	PyN	TF	sTF	PyT	DPyT	DTF
Raw (JPEG)	✓	✓	✓	✓	✓	✓
HDF5	✓	✓	✓	✓	✗	✗
Zarr	✓	✓	✓	✓	✗	✗
Pickle	✓	✓	✓	✓	✗	✗
NPY	✓	✓	✓	✓	✓	✓
NPZ	✓	✓	✓	✓	✗	✗
TFRs	✓	✓	✓	✓	✓	✓
TFRm	✓	✓	✓	✓	✗	✗

Table 1.5: Compatibility overview for ShapeNet data set

ShapeNet						
	PyN	TF	sTF	PyT	DPyT	DTF
Raw (OBJ meshes)	✓	✓	✓	✓	✗	✗
HDF5	✓	✓	✓	✓	✗	✗
Zarr	✓	✓	✓	✓	✗	✗
Pickle	✓	✓	✓	✓	✗	✗
NPY	✓	✓	✓	✓	✓	✓
NPZ	✓	✓	✓	✓	✗	✗
TFRs	✓	✗	✗	✓	✓	✓
TFRm	✓	✗	✗	✓	✗	✗

1.4 Terminology

Before beginning the thesis, it is important to clarify a few terms regarding benchmarking to avoid potential ambiguities, as these terms might not be used consistently across the community.

- Benchmark Suite: Under a benchmark suite (or just suite) we understand the process of benchmarking the different combinations of consumers, storage formats etc., for a given data set. Such a benchmark suite is made up of one or more individual benchmark runs.
- Benchmark Run: A benchmark run (or just run) is the process of benchmarking a single combination of parameters. In our case, the main parameters are the used consumer and storage format. The different runs then compose an entire benchmark suite, for which we want to compare the results of the different runs.
- Rounds: Every benchmark run is repeated multiple times. We call these repetitions rounds. The runs are repeated to obtain more reliable results than through executing them once.

Further, we adopt the distinction between "conventional consumers" and "DALI consumers". Both consist of classes consuming a data set conversion. The conventional consumers build upon the classic principle of having a data set class and a data loader using this data set class to load the samples. The data set classes and data loaders are commonly implemented using a DL library such as PyTorch [16] or TensorFlow [15].

The DALI consumers are part of the NVIDIA DALI framework [17]. These employ pipelines [24], which bundle the DSI process. They contain readers [25], directly reading the data from disk, as well as potential further preprocessing functions [26].

Due to the fundamental differences in the structure of conventional and DALI consumers, we see the need to make this distinction to avoid potential confusion.

1.5 Structure of the Thesis

In this thesis, we first provide an overview of the DSI pipeline (Section 2.1) before discussing the framework structure and the individual components of the tool in depth (Section 2.2).

After outlining the design, we dive deeper into the currently implemented features of DSI-Bench (Section 2.3), how to extend and adapt it to the user's needs (Section 2.4), and how to use the benchmark (Section 3.1).

Subsequently, we will cover exemplary results generated by DSI-Bench (Chapter 4). This includes a subsection for the results of each individual data set implementation and a summary of general observations. The results are followed by a discussion (Chapter 5).

Lastly, we will cover the limitations of our framework (Chapter 6) and the possibilities for further work (Chapter 7).

Chapter 2

Framework Description

We begin this chapter by describing the DSI pipeline and outlining the structure of DSI-Bench, followed by a more detailed explanation of the implemented features. These features include the supported data sets, storage formats, and data loaders.

At the end of the chapter, we explain how to extend the benchmark. This enables the user to adapt the benchmark to their needs if the out-of-the-box features do not suffice.

2.1 The DSI Pipeline

Fig. 2.1 demonstrates the DSI pipeline for a single benchmark round. The process starts with a converted data set saved on disk. The conversion is determined by the individual data set and storage format of the run containing that round.

The individual samples of the data set are then provided to the consumers by a storage-format-specific data set class or a DALI pipeline, depending on what type of consumer is used in the current run. If the consumer is conventional, a storage-format-specific data set class is needed. The DALI consumers require DALI pipelines to access the data.

The consumers ingest the entire converted data set randomly during the round. In the last step of the DSI pipeline, the individual samples are either loaded to the GPU memory or the CPU RAM. The chosen device depends on whether a GPU is available the system and the settings for the current benchmark execution.

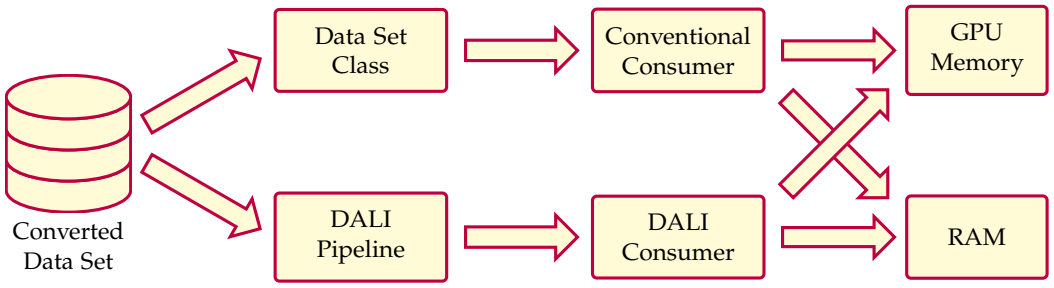


Figure 2.1: Diagram of the DSI process

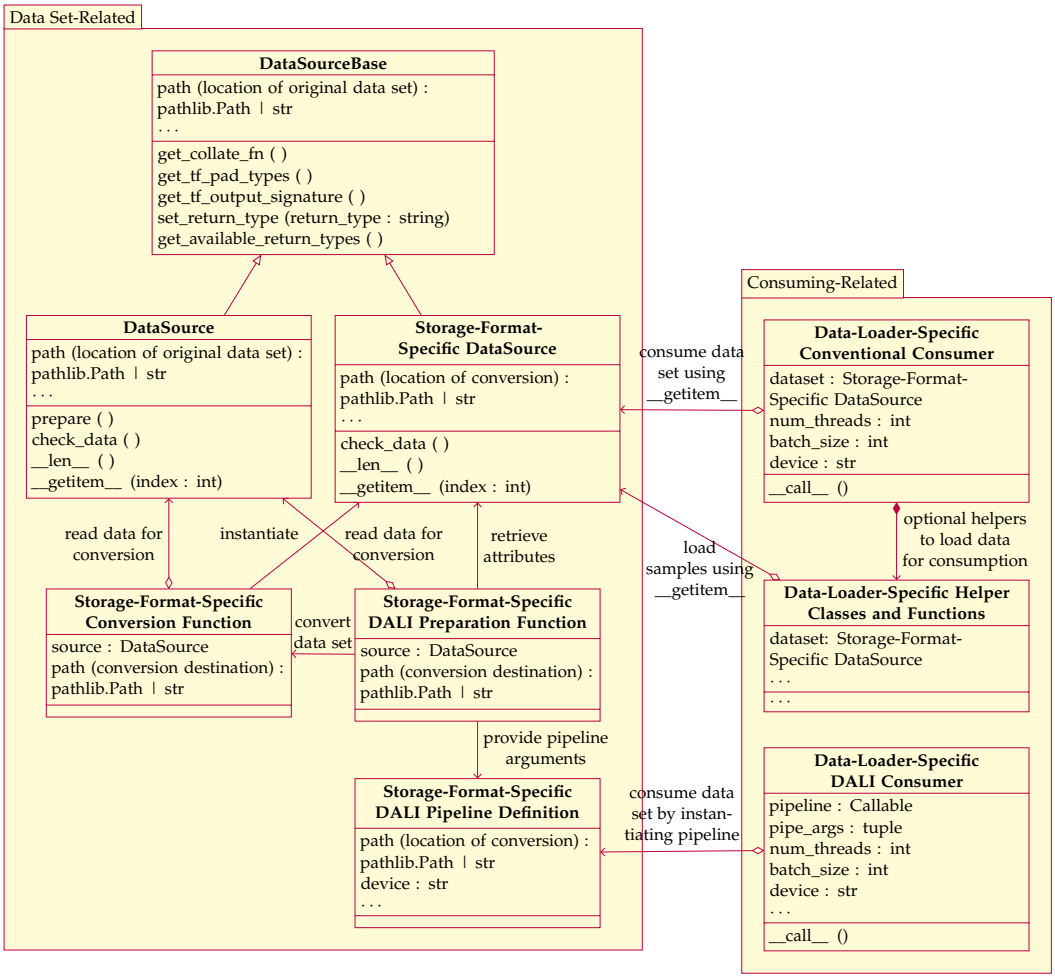


Figure 2.2: UML-style diagram of framework structure

2.2 Structure of the Benchmarking Framework

2.2.1 Overview

Fig. 2.2 illustrates the main components of our proposed framework DSI-Bench. Beware that Fig. 2.2 is more UML-style-oriented than formal UML. Some elements are standalone functions which do not belong to a class. Standalone functions cannot be modeled in classic UML and are drawn as classes in Fig. 2.2. They are named accordingly to allow for a clear distinction.

The symbol "· · ·" may appear in the attributes and methods of classes. This symbol indicates, that depending on the concrete storage format or data loader, there might be further elements. These are irrelevant for a general understanding of the structure and have been omitted from the diagram for a better overview, just as constructors and not implemented functions.

DSI-bench consists of two main areas. The first area contains the data set-related classes and functions, which provide the actual data. The second area focuses on loading and consuming the data. The members of the first area exist once for every implemented data set, which are a test data set, OADAT [6], ICON [7], ILSVRC 2012

[8] and ShapeNet [9]. OADAT is a data set consisting of synthetic opto-acoustic clinical data [6], ICON contains weather and climate simulation model data [7], ImageNet has labeled and categorized images [8], and ShapeNet is a data set of labeled and categorized meshes [9].

`DataSourceBase` is a base class inherited by the other data set classes. It is not meant for direct use but provides functions that the concrete implementations share. The class `DataSource` reads from the original data set provided in the suite. DataSource retrieves data for conversion to specific storage formats. The storage formats implemented in DSI-Bench are the original format of the data sets, HDF5 [10], Zarr [27], Pickle [12], NPY [13], NPZ [13], as well as two formats based on TFRecord files [14].

The storage-format-specific conversion functions take a `DataSource` object to read the original data and convert it to the according format if the conversion is not already present. Afterwards, the function returns a storage-format-specific data source object. These provide the functionality to read data from that data set saved in that specific format.

The storage-format-specific data source objects provide data for the conventional consumers. Two main components supply the DALI [17] consumers with data.The storage-format-specific DALI preparation functions ensure the data set conversion is present and return the arguments required by the corresponding storage-format-specific DALI pipeline definitions [24]. The pipeline definitions specify the steps of the DSI pipeline, such as reading and potentially padding the data.

The second main area of the framework is composed of classes and functions to load and consume the data. The main parts are the data-loader-specific conventional and DALI consumers. The conventional consumers implement data loading and consuming based on libraries such as TensorFlow [15] or PyTorch [16], as well as native Python. The conventional consumers may use data set wrappers and data loaders from the mentioned libraries.

The data loading can be offloaded to helpers, allowing for cleaner code. The helpers are instantiated by the consumers and provide the samples for consumption. The data is provided by a storage format-specific data sources.

The DALI consumers do not load data from one of the data source classes. The DALI consumers take a pipeline definition and the arguments required by it as constructor arguments. The pipeline arguments are generated by the DALI preparation functions. The pipeline is then instantiated and used to ingest the samples of the corresponding conversion.

2.2.2 Benchmark Script

There is a benchmark script acting as the entry point to the framework. It ties the main components of the benchmark together and applies the different configuration options. During execution, the script takes care of passing the correct arguments to functions and constructors, as well as makes sure that the data sets are converted accordingly. The user does not need to worry about such details, simplifying the use of DSI-Bench.

When executing the benchmark, the script iterates through the individual selected benchmark suites as well as the specified storage formats and data loaders. These different options result in the concrete runs, which are executed for a specified amount of rounds.

2.2.3 Consumers

The consuming-related classes and functions are all located in a single file. The related structures, namely the data-loader-specific conventional and DALI consumers, as well as optional helper functions and classes (Fig. 2.2), exist for various data loaders. The benchmark script selects the consumers to load and consume the various data sets based on the chosen settings.

For the description of the framework structure, we differentiate between conventional and DALI consumers, since they follow different designs. Conventional consumers are based on some data set class or wrapper possibly provided by a DL framework. The data is then loaded through a data loader implementation, taking care of accessing and batching the samples provided by the data set class. The data loading implementations may be offloaded to separate helper classes and functions, as illustrated in Fig. 2.2.

The DALI consumers do not require such data set classes or data loaders, but a DALI pipeline definition [24]. The DALI consumers instantiate the pipeline and access the data set through a reader function [25] used in the pipeline. This makes the need for a dedicated data set class superfluous. Since the pipeline definitions are heavily dependent on the data sets and not the consumers, they are located with the data set-related files.

Implementations of conventional consumers exist for a Python consumer without the use of other DL frameworks, a PyTorch [16] consumer, a TensorFlow [15] consumer, as well as a sequential TensorFlow consumer. There are two DALI consumer implementations in this project. One makes use of the DALI PyTorch plugin [28] and the other uses the DALI TensorFlow plugin [29].

During the data set consumption, the `device` setting may affect the behavior of the consumer. If a tensor is returned during consumption, the setting determines if the tensor is loaded into CPU RAM or into the VRAM of the GPU, if one is available.

There are certain parameters, which all consumers take in their constructor. These will be outlined here. The user does not need to be concerned with passing the correct arguments to the consumers, but they are still mentioned here to deepen the understanding of the framework.

- `num_threads`: The amount of threads used to parallelize data loading.
- `batch_size`: The batch size in which the samples should be loaded.
- `device`: An argument required by all consumers, but only affects those, which consume samples in the form of tensors. In this case, the tensor is loaded onto the according device during consumption. If no GPU is available, the behavior will default to loading into CPU RAM, no matter the `device` value.

The conventional consumers further take the argument `dataset`. This is a storage-format-specific data source instance, providing samples of the data set conversion used in the current run.

The DALI consumers require two further arguments.

- `pipeline`: The pipeline definition used to instantiate the pipeline ingesting the data.
- `pipe_args`: A tuple of arguments required by the pipeline definition. The arguments were previously generated by the preparation function corresponding to `pipeline`.

2.2.4 Data Set Classes

The different data set-related classes and functions illustrated in Fig. 2.2 exist once for every implemented data set. The code required for a single data set is grouped into its own file. The file contains the necessary code to convert the original data set into the different storage formats, read from the conversions, and supply data for the conventional and DALI consumers.

A definition for a class called `DataSourceBase` must be present in all data set files. It is a class inherited by other classes implementing actual data loading and is not intended for direct use. This enables us to have a base class for all storage-format-specific data source classes. The `DataSourceBase` must contain the following functions, some of which may raise a `NotImplementedError`, due to the class not being used directly:

- `__init__(self, path: pathlib.Path | str, ...)`: The constructor of the class. It must take an argument `path` pointing to the base path of the data set. It may contain other arguments specific to the data set. All constructors of all child classes must have the same arguments.
- `__len__(self)`: This returns the length of the data set used for benchmarking. It may be smaller than the original data set if only a subset is used for the benchmark. The function may raise a `NotImplementedError`.
- `__getitem__(self, index: int)`: This function returns the corresponding element of the data set. The function may raise a `NotImplementedError`.
- `check_data(self)`: This function checks if the data set located at `path` contains all necessary data. If not, an Error must be raised, indicating that the data set is incomplete, initiating a new conversion. The function may raise a `NotImplementedError`.
- `get_collate_fn(self)`: This function is needed to enable batching for the PyTorch data loader if not all elements returned by `__getitem__` have the same dimension. If they do, this function may return `None` [30]. Otherwise, it must return a `Callable` [31], which can be passed to the argument `collate_fn` of `torch.utils.data.DataLoader` [32]. To properly work with the rest of the framework, the elements of the collated batches need to support the PyTorch function `torch.Tensor.to` [33].

- `get_tf_pad_types(self)`: This function has the same purpose as `get_collate_fn`, but for the TensorFlow data loader. It may return `None` [30] if all items returned by `__getitem__` have the same dimension. Otherwise, it must return an element, which can be passed to `padded_shapes` of `tf.data.Dataset.padded_batch` [34].
- `get_tf_output_signature(self)`: This function must return an element of type `tf.TensorSpec` [35], specifying the shape and type of the returned samples. The returned element is needed for the TensorFlow data loader and passed to the `output_signature` argument of `tf.data.Dataset.from_generator` [36]. Since this function is inherited by the other data set classes, the returned samples of every conversion must have the same output signature.
- `set_return_type(self, return_type: str)`: This function sets the return type of the data set. The return type specifies if the function `__getitem__` returns the sample as a native Python structure (e.g. a `tuple`), a PyTorch or TensorFlow tensor. Be aware that the PyTorch data loader is not usable if the returned type is a TensorFlow tensor. The benchmark script handles this case for the user. An error is raised if the return type does not match a valid option.
- `get_available_return_types(self)`: This function returns a list with the three available return type options.

As illustrated in Fig. 2.2, the class `DataSource` inherits from `DataSourceBase`. It is designed to read the original data set located at `path` and must implement all functions, which raise a `NotImplementedError` in `DataSourceBase`. These are `check_data`, `__len__` and `__getitem__`. The implementation of `__getitem__` must not take the return type into account, since this class is only used to retrieve data for the conversion into other storage formats. It always behaves as if the return type were set to `python`. `DataSource` must also implement an additional function `prepare`. This function is called after the initialization of the `DataSource` object, before the actual benchmark or conversions. `prepare` can be used in case any preparation needs to be done for the data set.

For every implemented storage format there is an individual class inheriting from `DataSourceBase`. They enable reading data from the converted data set. The `path` argument from the constructor references the path to the conversion, not the original data set. The benchmark script handles passing the correct path for the user. For increased comparability, it is required that the samples returned by the `__getitem__` function of all storage-format-specific data sources have the same output signature.

There must be a conversion function for every storage format. The conversion function takes the arguments `source` and `output_path`. The elements of `source` are converted and saved to `output_path`. The conversion function returns an object of the corresponding storage-format-specific data set class. If the data set is already converted, it will not be converted again, given it passes the `check_data` function of the data set class.

Each data set file must contain a function `get_available_converters`, so the benchmark script knows, what storage formats are implemented. The function

returns a dictionary mapping the names of the implemented storage formats to the corresponding conversion functions.

DALI only supports a limited number of storage formats [17], each of which has its own reader [25]. This limits the amount of possible pipeline definitions. We implemented a pipeline definition for each by DALI and DSI-bench supported storage format.

For every pipeline, there must also be a preparation function. The purpose of the preparation function is to make sure, the data set is converted appropriately. Further, it returns all necessary arguments, required by the corresponding pipeline definition. The benchmark script takes care of passing the returned values of the preparation function to the DALI consumers. Every pipeline preparation function must take the following two arguments:

1. `source`: The `DataSource` instance used for conversion.
2. `output_path`: The path, where the converted data set is saved to.

These are the same arguments as the conversion functions take. They are called in the preparation functions.

Analogously to `get_available_converters`, there is a `get_available_pipelines` function. This returns a mapping from storage format names to another map with the following `key: value` pairs.

- `preparation`: Preparation function for the storage format.
- `pipeline`: Pipeline definition for the storage format.
- `labels`: The labels corresponding to the features returned by the pipeline. These are needed for the PyTorch DALI consumer to determine the length of the data set. It may contain the value `None` if the corresponding `DataSource-Base` has an attribute `key` naming the individual features. These will be used as labels if no explicit labels are provided.
- `pytorch_args`: A mapping with additional arguments only required by the DALI PyTorch consumer.
- `tensorflow_args`: A mapping with additional arguments only needed by the DALI TensorFlow consumer.

The function `get_available_pipelines` must take a list of keys as an argument if the classes of the data set have an equivalent attribute. This is needed since `labels` and certain arguments in `tensorflow_args` may depend on them.

2.2.5 `SysLoadBench`

The benchmark suites are executed using a package called `SysLoadBench` [37]. This package was created in the realm of this project since there was no Python package collecting all metrics in the format they were needed in. A separate package was created as opposed to directly integrating it into the benchmarking framework. `SysLoadBench` is more general and can be used to report the system load when executing any function and thus should be available by itself.

The metrics are gathered by creating a separate Python process for the rounds of every run, enabling more accurate results. `SysLoadBench` provides an untracked setup function, which is used to convert the data sets. The conversion therefore is not part of the benchmark result. Only actual consumption is measured. The CPU load and RAM utilization of the process executing the data set consumption and all its child processes are sampled every 0.05s.

For more details on `SysLoadBench` as a framework to gather system metrics, please refer to the official GitHub repository [37].

2.3 Features of the Proposed Framework

In this section, we go through and describe the features implemented in DSI-Bench. First, the storage formats supported by the data set implementations, the different consumers, as well as the data sets are covered, followed by the evaluated metrics.

2.3.1 Storage Formats

The following list of storage formats is implemented for every data set. Therefore, every implemented data set can be converted to and read from the storage formats listed below.

- `raw`: This is the original storage format of the data set. The subset used for benchmarking is copied to the same location as the other conversions, eliminating potential speed differences due to different drives. Unlike `DataSource`, the corresponding `DataSourceRaw` needs to take the `return_type` into account in `__getitem__`.
- `hdf5`: The subset of the data set used in the benchmark is saved in a single HDF5 [10] file. The Python package `h5py` [38] is used for the conversion as well as for reading the data.
- `zarr`: The data is stored in a Zarr [27] file. The Python package `zarr` [11] is used to interact with the Zarr files.
- `pickle`: The data gets pickled using `pickle.HIGHEST_PROTOCOL` [39] and every sample is saved into an individual pickle file [12].
- `npy`: The data is saved to NPY files [13] using `np.save` [40] and read again using `np.load` [41]. If the sample has multiple features, the features are each stored in individual NPY files, which are grouped in subdirectories. This directory structure facilitates the handling of the DALI pipelines using `npy`.
- `npz`: Similar to `npy` this is a NumPy storage format. In NPZ files multiple NPY files are saved in a zip file [13]. The data is stored in a compressed format using `np.savez_compressed` [42] and read using `np.load` [41].
- `tfrecord_single`: This is the first of two variants utilizing TFRecord files [14]. In both cases, the data is serialized before saving to avoid issues with the dimensions of the features. In this version, all samples are saved to one large TFRecord file.

- `tfrecord_multi`: This version is similar to the one above, but every sample is saved to an individual TFRecord file.

We decided to create two TFRecord storage formats for the following reason. We require shuffled data for consumption. Shuffled samples need random data access, but TFRecord files are designed for sequential access [43]. We were interested to see if it makes a difference to save all samples in a single TFRecord file (`tfrecord_single` format) and iterate through the data set up to the required sample, or to save all samples in individual files (`tfrecord_multi` format) and potentially have more disk accesses.

There are DALI pipeline definitions using the `tfrecord_single` and `npy` storage formats for every data set. Additionally, the ILSVRC 2012 data set has a pipeline utilizing the `raw` format, since the original images are saved as JPEGs, which can be read and decoded by DALI [44, 45].

2.3.2 Consumers

In this project, we developed 6 different consumers. In the following list, we refer to them by the command line interface (CLI) option, which they can be used with. The actual class name is mentioned in parenthesis following the option name to facilitate further references.

- `python` (`PythonConsumer`): This consumer loads and consumes the data without the use of other DL frameworks. Since our benchmark is rather I/O heavy than CPU heavy, we use python threads [46] rather than processes [47] to support parallelism. Due to the I/O heavy nature of the workload, Python threads are more efficient than processes [48].
- `tensorflow` (`TensorflowConsumer`): This consumer uses a data loader based on the TensorFlow data API [49]. A data set object [50] from the TensorFlow framework is created, which is consumed by iterating through it. To enable batching with different dimensions of individual samples it requires the `DataSourceBase` to define the `get_tf_pad_shapes` function. Further, the function `get_tf_output_types` needs to be defined and return the output signature of the returned samples.
- `tensorflow_sequence` (`TensorflowSequenceConsumer`): This consumer uses a data loader based on a sequence class [51] from TensorFlow's `keras` module [52]. It uses an ordered queue structure of `keras` [53] to consume the data.
- `pytorch` (`PytorchConsumer`): This consumer uses a PyTorch-based data loader [32]. To enable batching with different dimensions of individual samples it requires `DataSourceBase` to define the `get_collate_fn`.
- `dali_pytorch` (`DALIPytorchConsumer`): This consumer takes a NVIDIA DALI pipeline definition [24], as well as a tuple of arguments for it. It iterates through the data set using an iterator [54] provided by the DALI PyTorch plugin [28]. Besides the entry for `labels` in the dictionary returned by `get_available_pipelines`, no additional arguments are required in `pytorch_args`.

- `dali_tensorflow` (`DALITensorflowConsumer`): This consumer is similar to the DALI PyTorch consumer, but it uses an iterator [55] based on the DALI TensorFlow plugin [29] instead. Similar to the `TensorflowConsumer` this consumer requires some additional information about the shape and data type of the samples. This is why the additional arguments `shapes` and `dtypes` are required in `tensorflow_args`.

Since for DL training usually independent and identically distributed (i.i.d.) samples are required, all consumers load the data randomly. The indices of the data are either shuffled directly by the data loader or the list of indices was shuffled before going through that shuffled list in order.

This behavior is not guaranteed by the DALI consumers in the current implementation. We set the attribute `random_shuffle` [44, 56, 57] to `True` in the readers of our pipeline definitions. This causes the data to be read sequentially in batches of size `initial_fill` (1024 by default [44, 56, 57]) from where they are randomly sampled [44, 56, 57]. Therefore, the samples returned by the DALI consumers are not i.i.d.

2.3.3 Data Sets

During this thesis, we created implementations for 5 different data sets. In the following, they are mentioned using the name of the suite they are used in.

`TEST`

This is a data set to test the benchmark functions and to act as a reference for other data set implementations. The data consists of a NumPy array [58] with the entries `[1,2,...,10000]`. For the `raw` storage format, the samples are read directly from the in-memory NumPy array.

`OADAT`

This data set consists of synthetic clinical optoacoustic data [6] stored in an HDF5 file. Both `DataSourceRaw` and `DataSourceHDF5` read from HDF5 files, leading to the same expected performance as the implementations are the same for both classes.

Additional arguments of the constructor are:

- `filename`: The argument points to the actual HDF5 file relative to `path`.
- `key`: The argument contains the key(s) of the feature(s) in the HDF5 file, which should be used for benchmarking. Like this, the data can be reduced to make the benchmark smaller if desired.

`ICON`

This is climate simulation data [7] stored in an HDF5 file. Both `DataSourceRaw` and `DataSourceHDF5` read from HDF5 files.

Additional arguments of the constructor are:

- `filename`: This points to the actual HDF5 file relative `path`.

- `key`: This contains the key(s) of the feature(s) in the HDF5 file, which should be used for benchmarking. Like this, the data can be reduced to make the benchmark smaller if desired.

`ImageNet`

This is the data set of ILSVRC 2012. It consists of categorized JPEG images [8]. Our implementation only returns the image data and not the categories they belong to. Further, the images are resized to the dimensions $[\,224\ 224\ 3\,]$ prior to conversion to simplify batching for the different consumers.

Additional arguments of the constructor are:

- `amount`: This specifies the amount of images, which should be consumed.

The argument `path` needs to point directly to the validation data set or any other directory directly containing the JPEGs. They may not be located in subdirectories grouping them by category.

`ShapeNet`

We use ShapeNetCore (v2) [9], which consists of labeled meshes belonging to different categories. For the conventional consumers, the model and category ID, the label, the list of vertices, and the list of faces are returned. Due to technical limitations, only the lists of vertices and faces are returned by the DALI pipelines.

Additional arguments of the constructor are:

- `key`: This specifies the categories of the meshes, which should be used for benchmarking. This is not the category ID, but rather the human-readable name. The ID-name-mapping from [59] is used. If a category is empty or not present, it will be ignored. Otherwise, all models of that category are consumed.

2.3.4 Gathered Metrics

The following list of system metrics is gathered. The term in parenthesis represents the value corresponding to the later used placeholder `<metric>`.

- Runtime (`time`): The time needed to consume the data set.
- CPU Load (`cpu`): The CPU load, given in percent. 100% is the equivalent of one CPU core being fully utilized. Be aware that a percentage of e.g. 180% can also mean, that three cores are used with 50%, 60%, and 70%.
- RAM Utilization (`ram`): The amount of RAM used during execution.
- Disk Space (`used_diskspace`): The space used on disk by the different data set conversions.

Due to the use of the `setup` and `benchmark` function of `SysLoadBench`, only the consumption of the data set is measured [37]. The conversion happens in `setup` and therefore is not measured [37]. Before the actual consumption, garbage collection is

called to make sure, that there is no unnecessary data loaded in the RAM, inflating its utilization.

If subprocesses are created during the consumption of the data set, their CPU load and RAM utilization are also accounted for.

The results of each benchmark suite are saved in the directory `results`, where each suite has a subdirectory `<suite_name>`.

The unprocessed measurements of every suite are saved in a JSON file. The file contains some information about the system on which the benchmark ran. This information is found under the key `system_information` with the structure illustrated in Fig. 2.3.

```
system_information
├── python_version:  python version used
├── platform:  system platform
├── operating_system:  name of the installed OS
├── cpu:  model name of the CPU
├── gpu:  model name of the GPU or empty if none is present
└── ram:  amount of RAM available on the system
```

Figure 2.3: Structure of results in `system_information` in JSON result file

The disk space required by the converted data sets can be found under the key `used_diskspace`. The names of the storage formats (Section 2.3.1) are used as subkeys. Do not compare the values to the size of the original data set, but to the size of the entry `raw`, since only a subset of data may be converted. Hence, the used disk space might be considerably smaller than the total size of the original data set. The numbers given denote the size of the conversion in MiB.

The last key is `run_results`, containing the statistics regarding `time`, `cpu`, and `ram` of the individual runs and rounds. The results in `run_results` have the structure illustrated in Fig. 2.4.

There are graphs to compare the metrics visually. All the following paths are relative to the results directory of the corresponding benchmark suite. The graphs illustrate the mean values of the metrics and their standard deviations over the rounds of a run.

The graph `<suite_name>_diskspace.png` illustrates the metric `used_diskspace` (in MiB) for the different used storage formats.

The bar graphs `converter_graphs/<metric>/<converter_name>_<metric>_<device>_<return_type>.png` compare the metric `<metric>` between the different data loaders when `<converter_name>` is used as a storage format. If DALI data loaders are used, the results of their runs are added to all used return types to facilitate the comparison. If only DALI data loaders are used, `<return_type>=default` instead of one of the possible values.

The bar graphs `consumer_graphs/<metric>/<consumer>_<metric>_<device>_<return_type>.png` compare the metric `<metric>` between the different storage formats when `<consumer>` is used as a consumer. If DALI data loaders are used, the

results of their runs are added to the graphs for all used return types to facilitate comparison. If only DALI data loaders are used, `<return_type>=default` instead of one of the possible values.

The graphs `summary_graphs/<metric>/<metric>_<device>_<return_type>.png` are heat maps to compare the metric `<metric>` between all runs of a suite in a single graph. To increase readability the numbers are omitted. There is a logarithmic, labeled color scale. If DALI consumers are used, the results of their runs are added to the graphs for all used return types to facilitate comparison. If only DALI consumers are used, `<return_type>=default` instead of one of the possible values. For a more detailed comparison, there are labeled versions `summary_graphs/<metric>/<metric>_<device>_<return_type>_label.png` of the summary graphs.

- `run_results`
  - `<converter_name> <consumer_name> <device> <return_type>` represents a single run of a suite (due to the different structure of the DALI consumers `<return_type>` is omitted in these cases)
    - `<metric>`: either cpu (percent), ram (byte) or time (seconds)
      - $x \in [0, n-1]$ where n is the total amount of rounds executed for that run, holding the statistics of the corresponding individual (0-indexed) round (not available for the metric time, which only has one measurement per round)
        - `max`: maximum recorded value of metric in round
        - `mean`: mean value of metric in round
        - `stddev`: standard deviation of metriin round
        - `25`: 25th percentile of measurements
        - `50`: 50th percentile of measurements
        - `75`: 75th percentile of measurements
        - `90`: 90th percentile of measurements
        - `95`: 95th percentile of measurements
        - `99`: 99th percentile of measurements
      - `raw`: list of runtime measurements in seconds of the rounds
      - `total`: statistics of summarized over all rounds
        - `max`: maximum mean of all rounds
        - `mean`: mean of mean of rounds
        - `stddev`: standard deviation of the mean of the rounds, or over raw entries in case of metric time
        - `25`: 25th percentile of measurements
        - `50`: 50th percentile of measurements
        - `75`: 75th percentile of measurements
        - `90`: 90th percentile of measurements
        - `95`: 95th percentile of measurements
        - `99`: 99th percentile of measurements

Figure 2.4: Structure of results in `run_results` in JSON result file

2.4 Extending the Benchmark

One of the main goals of this thesis was to create a framework enabling data scientists to easily benchmark different DSI options for their own real-world data sets. To do this, the framework is designed in a modular way. In the following we cover how to add a custom implementation for data sets, consumers, as well as additional storage formats, should the provided ones (Section 2.3) not cover the user's needs.

2.4.1 Data Sets

Adding a custom data set requires to create e new file for the implementation. In this file, various classes and functions must and can be added. The required structure of the file is similar to that, of the existing implementations (Section 2.2.4) and is covered in this section. The new data set must be added to the benchmark script (Section 2.2.2), so it can be used when configuring a benchmark suite.

The content of the data set file should replicate the structure of the data set-related classes and functions visible in Fig. 2.2. Both the classes `DataSourceBase`, as well as `DataSource` must be present. Further, for any storage formats of interest, storage-format-specific conversion functions and `DataSources` must be implemented. The conversion functions can be used in DALI preparation functions. DALI pipeline definitions may be added, if runs with DALI consumers are relevant for the user.

In case a DALI pipeline definition is added, the readers used in the pipeline must be named, so the PyTorch DALI data loader can infer the length of the data set. If there is one reader per feature with corresponding `label[i]`, they need to be named `reader<label[i]>`. If there is only one reader it must be named `reader<label[0]>`.

Not all storage formats and pipeline definitions, which have been added to the project (Section 2.3.1), must be implemented. The user must add a DALI preparation function in case a DALI pipeline definition is added. The implemented storage formats with corresponding storage-format-specific conversion functions need to be provided in the function `get_available_converters` (Section 2.2.4) to be able to use them with the conventional consumers. Similarly, the added CALI-related functions must be added to `get_available_pipelines` (Section 2.2.4), so they can be used by the DALI consumers.

To be able to use the data set implementation in a suite configuration, it needs to be added to the benchmark script (Section 2.2.2). The data set file containing the classes and functions must be imported along with the other data set files. Afterwards, a unique name `<dataset_type>` must be chosen. When that is added to the data source lookup table `data_sources_lut`, `<dataset_type>` can be used to specify the new custom implementation when configuring a suite.

For further reference and a concrete example, a look at the implementation of the test data set [60] might prove to be useful.

2.4.2 Storage Formats

Adding a new storage format to an existing data set is similar to creating the required classes and functions for an existing storage format and a new data set. A new storage-format-specific conversion function and data source class need to be added to the relevant file. The conversion function must take an argument `source`, which is a `DataSource` object (Section 2.2.4) providing the necessary data. The second required argument is `output_path`, specifying where to save the data set conversion. The conversion function must return an instance of the newly specified data source class.

As with the existing storage formats, a unique name `<converter_name>` must be chosen for the format and added to `get_available_converters` alongside the corresponding conversion function. If a DALI pipeline definition using this format is added, the adequate entry must be added to `get_available_pipelines`. In case a DALI pipeline definition is added, a preparation function must be added as well.

To be able to use the new storage format when running the benchmark, the `<converter_name>` must be added to the list `default_converters` in the benchmark script (Section 2.2.2).

2.4.3 Consumers

Adding consumers is no more complicated than adding a new data set implementation (Section 2.4.1). We will outline the procedure in this section. The process entails adding a consumer class and potential helper classes and functions. Similar to adding a data set, the new consumer must be added to the benchmark script.

Since the design of the conventional consumers and DALI consumers differ substantially (Section 2.2.3), they are covered separately in this section.

Conventional Consumers

To add an implementation for a new conventional consumer, adding a consumer class to the consumer file (Section 2.2.3) is sufficient. It might be helpful to use additional data-loading classes or functions. This depends on the specific implementation and the frameworks used.

The structure of the new consumer class must be similar to that of the other conventional consumers (Section 2.2.3). It must take the same arguments. A `__call__` function must be implemented, which consumes the entire data set passed through the constructor. During the consumption, the consumer must respect the `device` setting. This specifies if the returned element is loaded into the RAM of the CPU or VRAM of the GPU.

As a last step, the consumer implementation must be added to the benchmark script (Section 2.2.2). A unique name `<consumer_name>` must be chosen, with that the consumer can be addressed when running the benchmark. `<consumer_name>` must be added to the list `default_consumers`. Lastly, the mapping `<consumer_name>: <class_name>` must be added to the consumer lookup table `consumer_lut`. `<class_name>` specifies the name of the new consumer class.

As a concrete reference for a conventional consumer, the `TensorflowSequenceConsumer` [61] might be useful. It employs a separate data loader `DataLoaderTensorflowSequence` to provide the data and nicely illustrates the outlined steps.

NVIDIA DALI Consumers

Adding a further NVIDIA DALI based data loader requires a few extra steps compared to adding a conventional data loader.

The additional arguments `pipeline` and `pipe_args` are required in the constructor. If further arguments are necessary, these can be added as a dictionary to the entries returned by the function `get_available_pipelines` (Section 2.2.4) of the individual data sets. The name of this dictionary can be chosen freely. If no additional arguments are required, this step is not necessary.

The chosen `<consumer_name>` must contain the term "dali" so that it is treated as a NVIDIA DALI consumer by the benchmark script. Otherwise, it will be treated as a conventional consumer. In case the DALI consumer requires additional arguments for the constructor, this must be handled in the definition of `setup_func` in the benchmark script (Sections 2.2.2 and 2.2.5). An extra case must be added to pass the right arguments of the `get_available_pipelines` entry to the constructor. If no further arguments are used in the constructor of the consumer, this is not necessary and everything is handled through the default case.

To facilitate the implementation, the DALI consumers `DALIPytorchConsumer` [62] and `DALITensorflowConsumer` [63] may provide helpful guidance.

Chapter 3

Use of the Framework

In this chapter, we cover how to use DSI-Bench. First, we describe how to set up the environment containing all required dependencies. Afterward, we go over the different configuration options for running the benchmark.

Before setting up the environment and configuring the benchmark, the first step is to clone the repository containing all the code as well as files facilitating the setup. The repository can be found at [5].

3.1 Environment

There are two provided ways to set up an environment to work with the benchmark. There is a standard conda [64] environment, as well as a Docker image [65]. The packages in the DSI-Bench framework itself are managed using Poetry [66].

One part of the environment is the path where the converted data sets are saved. The path is specified through the environment variable `TMP_DATA_DIRECTORY` and the default location is `./tmp`, relative to the project root. If not enough space is available on that drive, the user is required to change the location. Different disks might yield different results due to non-identical hardware.

3.1.1 Conda Environment

If the user prefers to set up a conda environment, conda first needs to be installed on the user's machine. The project contains a bash script `environment/install-env.sh` [67] executing all steps necessary to set up a conda environment with all required packages. The script also makes sure that the packages of the framework itself are registered through Poetry.

A string may be passed as an argument to the installation script. The string specifies the name of the created conda environment. If none is passed, the name `benchmark` is chosen.

3.1.2 Docker

Alternatively to installing a conda environment all dependencies and the code are bundled into a docker image, that can be found in the Docker Hub repository [68].

Having Docker images facilitates the deployment of the framework and its spread in the data science community.

To make sure the Docker image stays up to date, a GitHub Action [69] was defined to rebuild and push the image to the Docker Hub registry [68], whenever a change to the `main` branch of the project repository [5] is made.

There are three available tags for the image. The first tag is `latest`, which contains the latest version of the `main` branch of the GitHub repository, a conda installation, as well as some other packages for convenience. For more detailed information about the content of this tag, please refer to the description of the Docker Hub repository [68]. This is the version built and pushed by the GitHub Action. Due to size limitations of the GitHub pipeline, the conda environment is not prebuilt. Everything, that is necessary to build the environment as described in Section 3.1.1 is included in the image, such as conda and the installation script.

The second available tag is `standard`. It is equivalent to `latest`, but the dependencies are already built into a conda environment named `benchmark`. This makes the image too large to build it using GitHub actions. It is up to date at the time of the thesis' publication and can be built by the user for the newest version if necessary. The process is outlined in the `README.md` [70] of the project's GitHub repository.

The third tag is `runai`. This version is very similar to the tag `standard`. It is optimized for the use and development on the Run:ai platform of SDSC [71]. It sets the home directory of the user to `/myhome` and further activates SSH in the container. The user can connect with the running container over SSH to enable remote development. See the description in the Docker Hub repository [68] for more details on the use of this image version.

3.2 Configuring the Benchmark

There exist two main options to configure the benchmark. The first is YAML files. These configuration files specify the individual benchmark suites. The second option is command line interface (CLI) arguments when executing the benchmark script (Section 2.2.2). These are mainly used to specify the suites that should be executed, as well as the individual combinations of consumers and storage formats making up the runs of the suites.

3.2.1 Configuration Files

The YAML configuration files are grouped in the directory `./configs` relative to the project root. They define individual benchmark suites. The naming of the configuration files is irrelevant, as long as the file name is unique.

The files contain several key-value pairs. The following must be present in all configuration files:

- `suite_name`: The name given to the suite defined by this configuration. The name must be unique under all configuration files.
- `dataset_type`: The type of data set used for the suite. It refers to the according `<dataset_type>` (Section 2.4.1) of the data set implementations.

The implementation must be able to read the data set of this suite. For the implemented data sets, the options are `test`, `oadat`, `icon`, `imagenet`, and `shapenet`.

- `path`: The base path of the original data set used for this suite.
- `batch_size`: The batch size of the samples during loading.
- `num_threads`: The amount of threads used during loading.

The individual data source classes may have further parameters in their constructors. If a certain data set type is used, all additional parameters of the constructor must be added to the configuration file. The key must be equal to the parameter name.

If a new configuration file is added, it is not executed by default. The `<suite_name>` may be added to the list `default_suites` in the benchmark script (Section 2.2.2). Then it will be used as a default if no other suite is specified. Currently, there is one configuration file for every implemented data set (Section 2.3.3). Their `suite_names` are `TEST`, `OADAT`, `ICON`, `ImageNet`, `ShapeNet`.

3.2.2 CLI Options

When running the benchmark script `benchmarks/benchmarks.py` [72] there are several command line interface (CLI) options. They are used to specify which suites are executed, as well as the combination of storage format, consumers, device and return type, making up the runs of the individual suites. The following list explains the different options.

- `suites`: Following this option is a list of `suite_names`, which should be benchmarked. The current default if none are given is `TEST`, `OADAT`, `ICON`, `ImageNet`, and `ShapeNet` (Section 2.3.3).
- `converter_names`: The list following this option determines for which storage formats the data sets get benchmarked. The options `raw`, `hdf5`, `zarr`, `pickle`, `npy`, `npz`, `tfrecord_single`, and `tfrecord_multi` are possible. They represent the implemented storage formats (Section 2.3.1). If no value is given, the default is all possible.
- `consumers`: This parameter determines which of the consumers is used. The possible values are `python`, `tensorflow`, `tensorflow_sequence`, `pytorch`, `dali_pytorch`, and `dali_tensorflow` (Section 2.3.2). If no option is given, all will be used.
- `devices`: The possible options here are `cpu` and `gpu`, determining if the data should be loaded into the GPU memory during the consumption of the data set. If no option is given, both will be used. The option `gpu` is only available if there is a CUDA-compatible GPU [73] available on the system.
- `return_types`: The list following this option determines the type of the return value of the `__getitem__` function of the different data source classes (Section 2.2.4). The possible options are `python`, `tensorflow`, and `pytorch`. If none is given, all will be used.

- `name_prefix`: If a string is provided with this option, the name of the suite will be prefixed to `<name_prefix><suite_name>`.
- `rounds`: This integer must have a value in $x \in [1, 999]$. It determines the number of rounds for which each run is repeated. A higher number results in more reliable results, but also increases the runtime of the benchmark script. The current default is `3`.
- `delete_conversions` / `keep_conversions`: These flags enable or disable the deletion of the data set conversions. If `delete_conversions` is passed (default), the corresponding conversion in `TMP_DATA_DIRECTORY` is deleted after the benchmark suite has completed.

For every element of `suites` the corresponding suite will be executed. There is a run for every valid element of the cross product `converter_names` $\times$ `consumers` $\times$ `devices` $\times$ `return_types`. For the valid combinations of consumers and storage formats, please refer to Tables 1.1 to 1.5. Further, the combination `gpu` for `devices` and `python` for `consumers` is incompatible, since not necessarily a tensor is returned. In the case that a tuple or dictionary is returned, it cannot be loaded into the GPU memory.

Chapter 4

Benchmark Results

In the following, we discuss exemplary results generated by DSI-Bench. We benchmarked one configuration each for the OADAT, ICON, ILSVRC 2012, and ShapeNet data sets. The suite configurations (Section 3.2.1) can be found in the configuration files A.1 to A.4 in Appendix A.

The command used to run the benchmark script is `python benchmarks/benchmarks.py --suites OADAT ICON ImageNet ShapeNet --device=gpu`. It follows, that the default amount of rounds, `rounds=3`, was used for every run.

The benchmark was run in a containerized docker environment [68] on a GPU compute node of the SDSC. The node has 270 GB of RAM, two `AMD EPYC 7302P` Processors, as well as an `NVIDIA A100` GPU with 80 GB of VRAM, of which 15% was allocated to the benchmark. The node was shared during the benchmark, which might have influenced the runtime. For most research groups this shared node scenario should be more common than having an isolated node for a project, so this makes the results more realistic.

For legibility reasons, the graphs shown in this section do not contain any numerical values. Identical graphs containing the numerical results as well as the standard deviation can be found in Appendices A.1 to A.4. In the graphs the value of runs including DALI consumers are the same in each subplot of a figure (Section 2.3.4).

4.1 OADAT

For the benchmark using OADAT, a subset of 1000 samples of the file `SWFD_semicircle_RawPB.h5` was used. The subset was saved to an individual file. The chosen batch size was 32 and the number of threads used was 12 (configuration file A.1).

4.1.1 Used Disk Space

As can be seen in Fig. 4.1, the sizes of the different conversions all lie within roughly 10% of each other. The additional compression used in the `npz` format only makes a difference of 7% when compared with the uncompressed `npy` format.

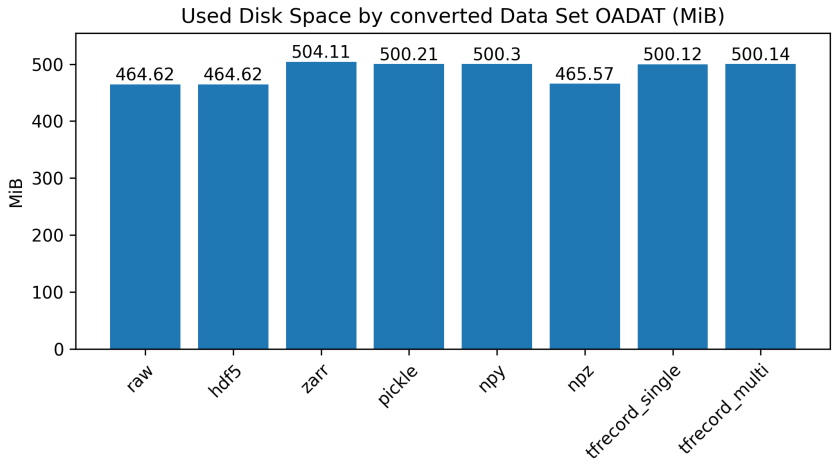
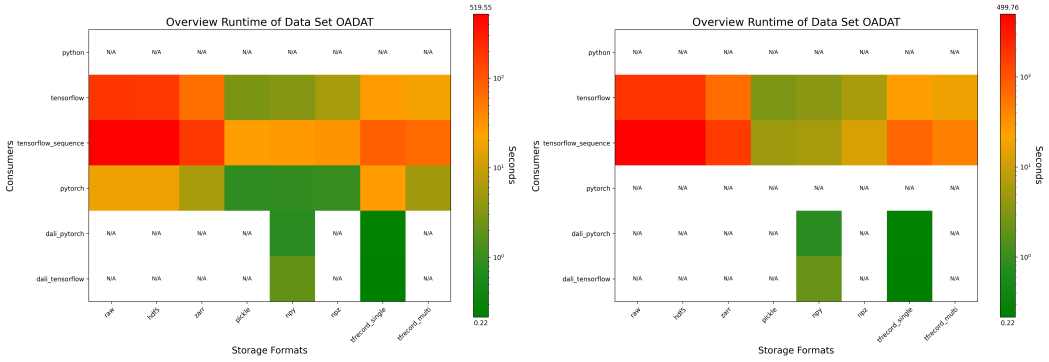
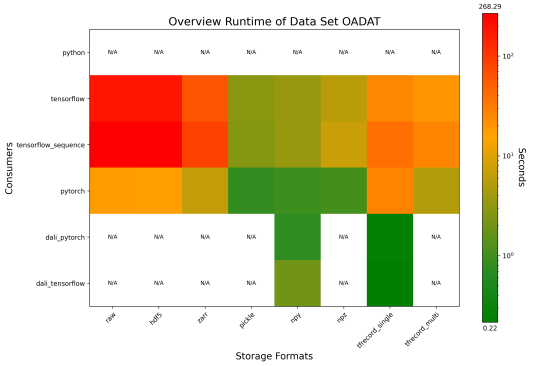


Figure 4.1: Disk space used by conversions of OADAT data set

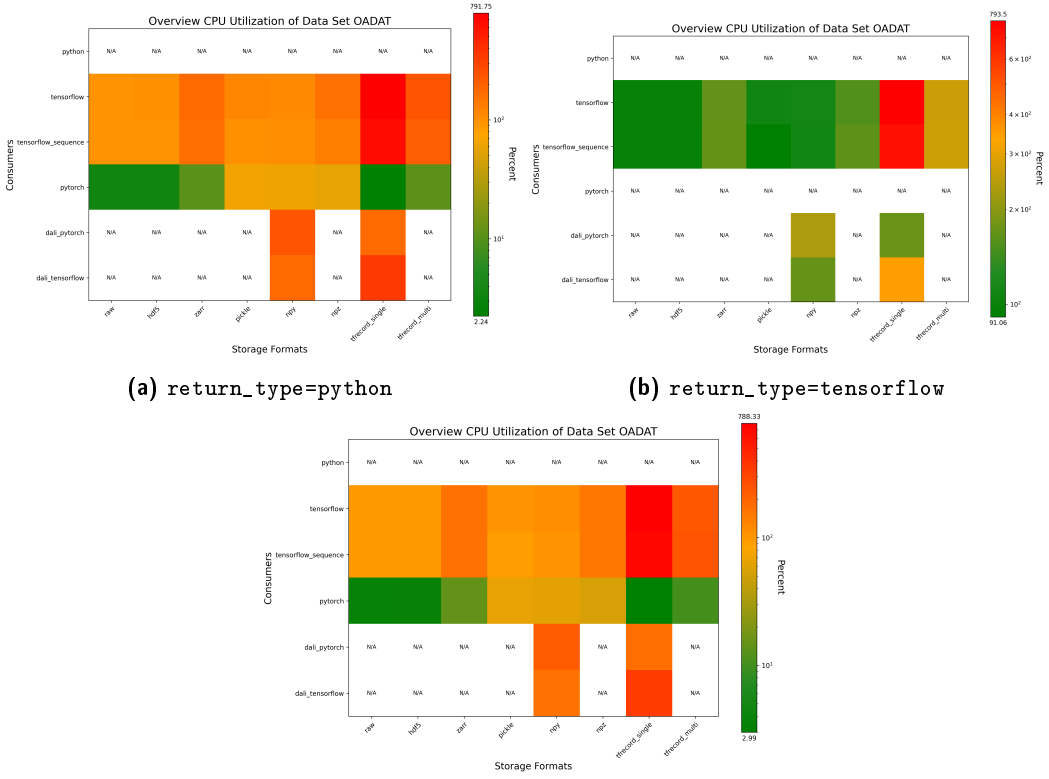


(a) `return_type=python` **(b)** `return_type=tensorflow`



(c) `return_type=pytorch`

Figure 4.2: Runtime results of OADAT



(a) `return_type=python` (b) `return_type=tensorflow`

(c) `return_type=pytorch`

Figure 4.3: CPU utilization results of OADAT

4.1.2 Runtime

Over all return types in Fig. 4.2, there are two general trends that can be seen regarding the runtime. Generally, the `raw/hdf5` and `zarr` storage formats are the slowest for this data set, although this is the original format and Zarr has a similar structure to HDF5.

The second observable trend is, that the NVIDIA DALI consumers load the data the fastest. For the conventional consumers, the storage formats `pickle`, `npy`, and `npz` are considerably faster than the rest, especially in combination with the PyTorch consumer. It is the fastest conventional consumer in this case, after which the TensorFlow and then the sequential TensorFlow consumer follow.

When comparing Fig. 4.2a to Fig. 4.2c or Fig. 4.2b, we can see that the sequential TensorFlow consumer profits from a return type already returning a tensor. If `return_type=python`, the run times are significantly slower than using one of the other options. The sequential TensorFlow consumer is fastest if `return_type=pytorch`.

For the other consumers, the return type does not make a significant difference when loading the OADAT data set.

4.1.3 CPU Utilization

The average CPU utilization in general does not differ very much when comparing the results of the different return types in Figs. 4.3a to 4.3c with each other.

Considerably more CPU was utilized to read the data stored in the `tfrecord_single` storage format combined with the TensorFlow or sequential TensorFlow consumers. Contrarily, the PyTorch consumer shows the lowest CPU utilization out of all storage formats for this option. It shows the highest average CPU utilization for the formats `pickle`, `npy`, and `npz`, which may be caused by the effort required to deserialize and/or decompress the stored samples. Another cause could be the lower runtime compared to the other storage formats causing an increased average.

Another interesting observation is, that the PyTorch consumer generally does not use a lot of CPU compared with the other consumers. This could have different causes. Possibly, when creating the PyTorch data set, a larger amount of the data is prefetched. This would reduce the need for CPU compute during the start of the consumption.

The DALI consumers on the other hand have a higher average CPU utilization. Unlike the conventional counterparts, there does not seem to be a large difference between the DALI PyTorch and DALI TensorFlow consumers.

Please be aware, that although the TensorFlow and sequential TensorFlow consumers seem to be using a lot less CPU for `return_type=tensorflow` in Fig. 4.3b, compared to Figs. 4.3a and 4.3c. This is deceiving. The PyTorch consumer showed a significantly lower CPU utilization than the other consumers. Since it was omitted for those runs (Section 2.2.4), the increased minimum value caused a shift in the color scale.

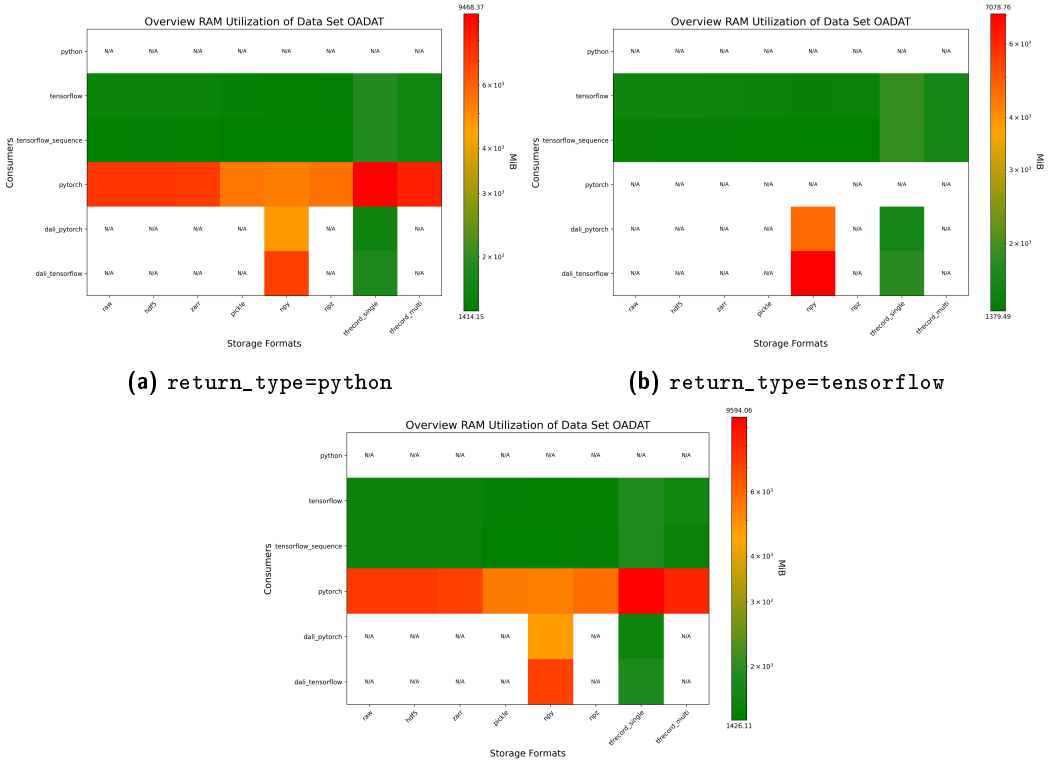
4.1.4 RAM Utilization

As can be seen by comparing Figs. 4.4a to 4.4c with each other, the RAM utilization does not depend much on the return type.

It is notable, that the PyTorch consumer seems to be using roughly three to five times the amount of RAM the other conventional consumers use. This supports the idea, that PyTorch does more prefetching when creating the PyTorch data set. If RAM usage is a concern, this should be kept in mind.

The average RAM utilization for the PyTorch consumer is lowest for the storage formats `pickle`, `npy`, and `npz`, where it has shown the highest CPU utilization. This seems to be a trend among all conventional consumers.

For the DALI consumers, we can observe the opposite behavior, where the runs with `npy` use more RAM than those with `tfrecord_single`. Further, the DALI TensorFlow consumer uses more RAM than the DALI PyTorch consumer for both implemented pipelines.



(a) `return_type=python` **(b)** `return_type=tensorflow`

(c) `return_type=pytorch`

Figure 4.4: RAM utilization results of OADAT

4.2 ICON

For this suite, a subset of 10000 samples of the file `train_10p.h5` was used. The subset was saved to an individual file. The chosen batch size was 32 and the number of threads used was 12 (configuration file A.2).

Due to technical difficulties, results for runs combining the TensorFlow or sequential TensorFlow consumers with the storage formats `tfrecord_single` or `tfrecord_multi` could not be executed on the GPU node. They were omitted.

4.2.1 Used Disk Space

In Fig. 4.5 we can observe that every storage format, except `hdf5/raw`, used at least double the disk space required by `raw`. This also holds for formats utilizing compression, such as `npz`, or formats where the samples are serialized, such as `tfrecord_single` and `tfrecord_multi`.

The used samples of ICON consist of features, which are either scalars or of dimension (70) or (71). In contrast, all features of OADAT have a dimension of $(256, 256)$. Some storage formats, like `pickle` or `npy`, store individual samples to their own files or serialize them. Serializing entails saving information about the structure of a sample alongside the actual data. Thus, if a feature is small, the overhead of serializing or additional files becomes proportionally larger compared

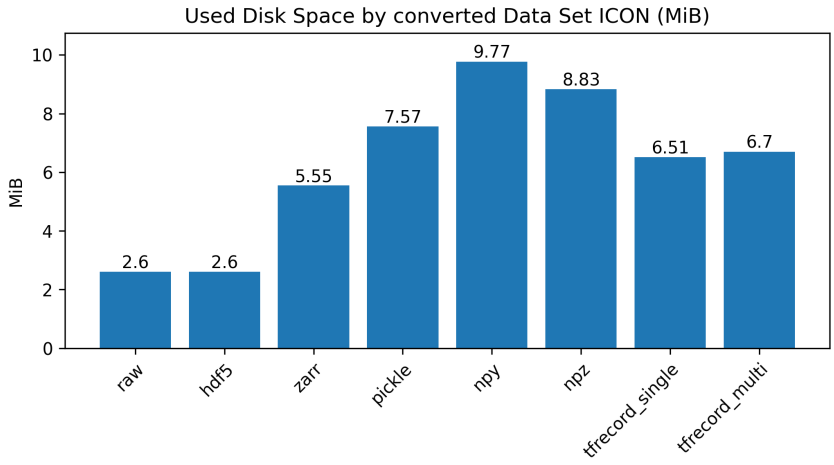
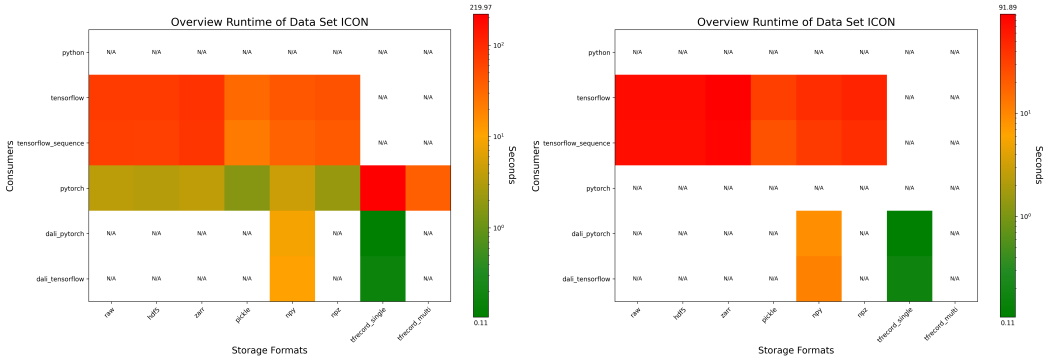
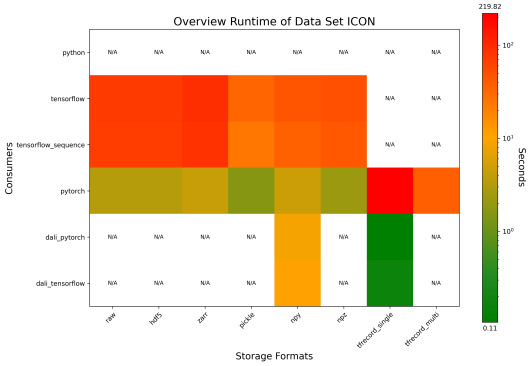


Figure 4.5: Disk space used by conversions of ICON data set



(a) `return_type=python` **(b)** `return_type=tensorflow`



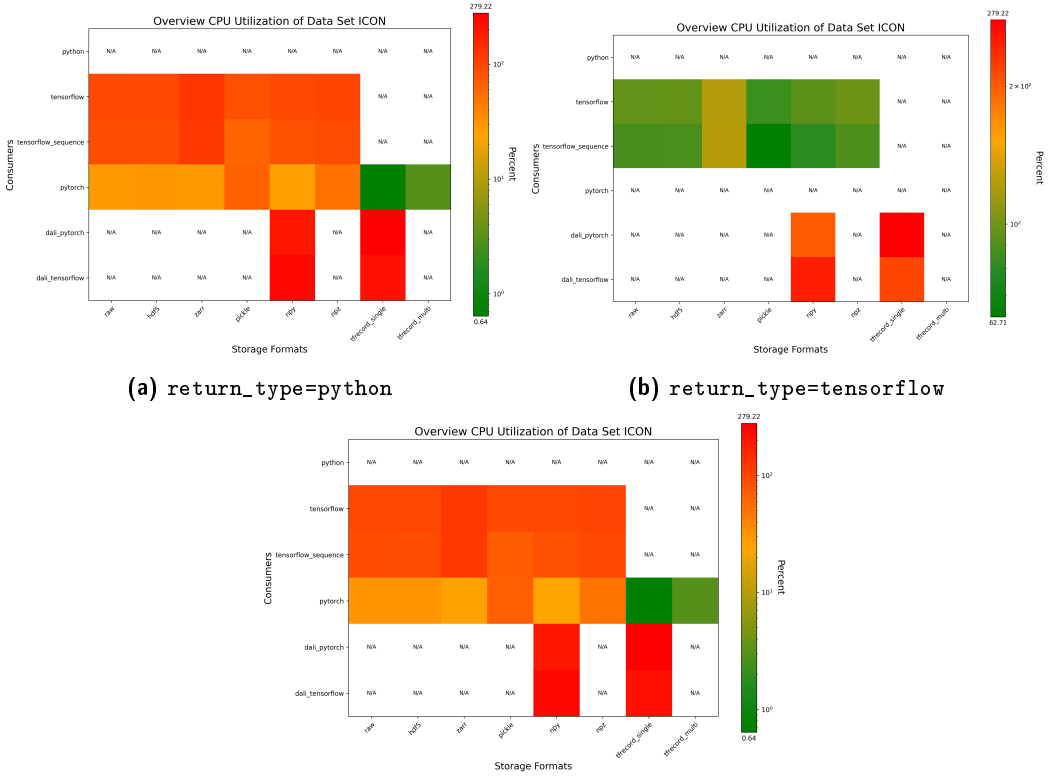
(c) `return_type=pytorch`

Figure 4.6: Runtime results of ICON

to the amount of data. Since ICON has smaller features, the effect becomes more evident (Fig. 4.5) compared to OADAT (Fig. 4.1).

4.2.2 Runtime

Comparing Fig. 4.6a to Figs. 4.6b and 4.6c we see that `return_type=python` is slightly faster for most runs using the conventional data loaders. The difference



(a) `return_type=python` (b) `return_type=tensorflow`

(c) `return_type=pytorch`

Figure 4.7: CPU utilization results of ICON

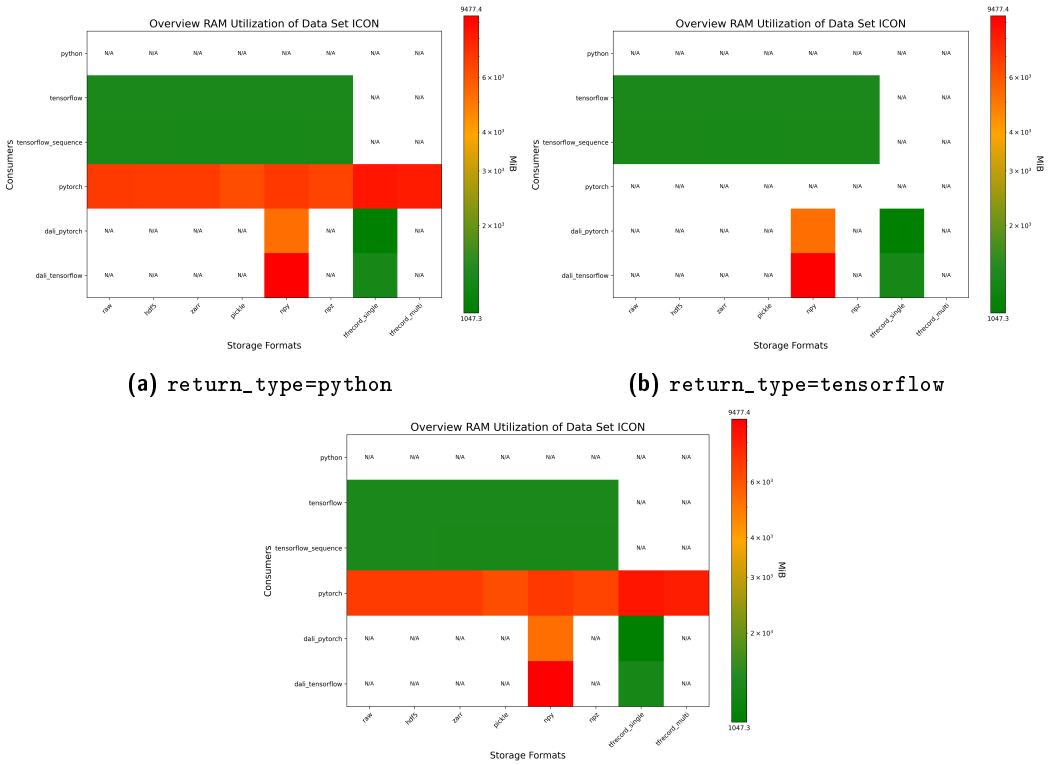
is much less significant than with the sequential TensorFlow consumer and the OADAT data set.

As can be seen in Figs. 4.6a and 4.6c the TFRecord file formats are the slowest storage formats for the PyTorch consumer. For the DALI consumers, we can observe the opposite, where the `tfrecord_single` pipeline performs a lot better than the `npy` pipeline.

The difference could be explained by the larger number of individual samples and smaller individual features, causing more single file reads by the `npy` pipeline, each consuming fewer data. The larger amount of file accesses could cause a proportionally larger overhead compared to the `tfrecord_single` pipeline, where all samples are saved in one file and thus potentially need less accesses to disk.

For the conventional consumers, the storage format `pickle` seems to be the fastest overall. Comparing the conventional consumers with each other in Fig. 4.6 we can further state that the PyTorch consumer is the fastest for all available storage formats.

For the ICON data set the DALI TensorFlow consumer was slower than the DALI PyTorch consumer with both pipelines.



(a) `return_type=python` (b) `return_type=tensorflow`

(c) `return_type=pytorch`

Figure 4.8: RAM utilization results of ICON

4.2.3 CPU Utilization

The CPU utilization is not influenced by the return type in general. Comparing Fig. 4.7c with Figs. 4.7a and 4.7b we can see an exception to this. When using the return type `pytorch` the TensorFlow and sequential TensorFlow consumers experience a higher average CPU utilization for most runs.

The PyTorch consumer does use less CPU than the other consumers in most runs. For the `pickle` storage format, the conventional consumers experience a similar CPU utilization. The PyTorch consumer shows an especially low CPU load for the storage formats using TFRecords.

In contrast to the PyTorch consumer, the DALI consumers require the most CPU compute, about double compared to the next conventional consumer. Which DALI consumer has a higher CPU utilization depends on the storage format.

Note, that the TensorFlow and sequential TensorFlow consumers seem to be using a lot less CPU in Fig. 4.7b, compared to Figs. 4.7a and 4.7c. This only looks this way due to a shift in the color scale caused by the absence of the PyTorch consumer.

4.2.4 RAM Utilization

Comparing Figs. 4.8a to 4.8c with each other, there is a significant difference in average RAM utilization across the runs with different return types.

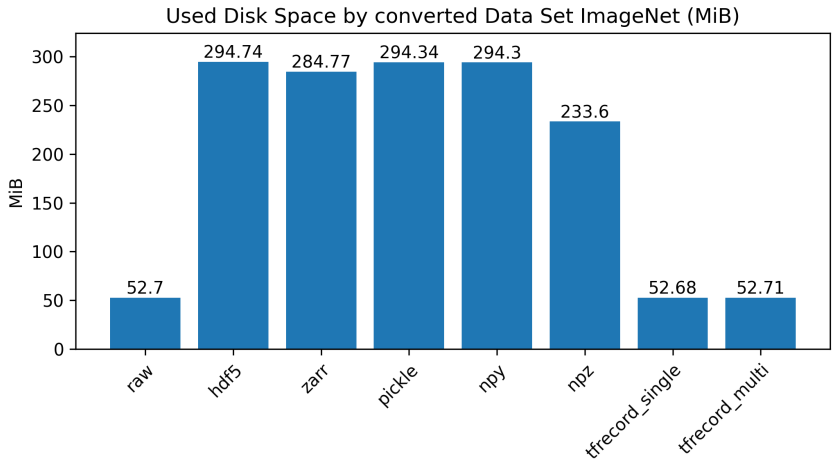


Figure 4.9: Disk space used by conversions of ILSVRC 2012 data set

In Figs. 4.8a and 4.8c, we can observe that the PyTorch consumer uses about five times more RAM than the other conventional consumers when comparing the runs for a fixed storage format. If the PyTorch consumer does more prefetching than the other conventional consumers, this could explain the elevated RAM usage.

Other than that, the RAM usage of the conventional consumers seems consistent across storage formats, with the exception that the PyTorch consumer uses roughly 25-35% more RAM when paired with the TFRecord storage formats compared to the others.

Out of all combinations, the DALI TensorFlow consumer combined with the `npy` pipeline uses the most RAM. The RAM utilization is about 35% higher than that of the PyTorch consumer paired with `npy`. The RAM usage of the DALI PyTorch consumer is elevated as well compared to the TensorFlow and sequential TensorFlow consumers. It is still lower than that of the PyTorch consumer and its DALI counterpart.

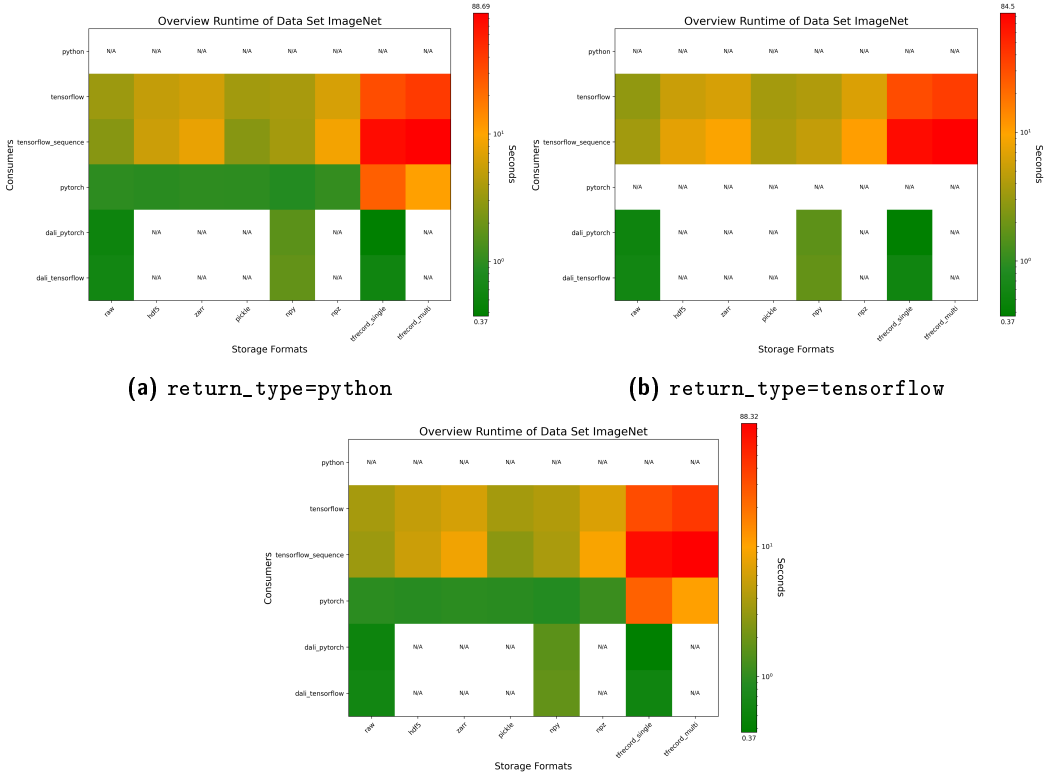
For the `tensorflow_single` file format, the DALI consumers both show a low average RAM utilization. The combination between the DALI PyTorch consumer and `tfrecord_single` shows the lowest observed RAM utilization out of all runs for this suite.

4.3 ILSVRC 2012

For the benchmark, a subset of 2048 images from the validation set was used. The chosen batch size was 32 and the number of threads used was 12 (configuration file A.3).

4.3.1 Used Disk Space

As seen in Fig. 4.9, the original storage format uses considerably less space than the options `hdf5`, `zarr`, `pickle`, `npy`, and `npz`. This could be caused by the fact that the



(a) `return_type=python`

(b) `return_type=tensorflow`

(c) `return_type=pytorch`

Figure 4.10: Runtime results of ILSVRC 2012

original images are stored as JPEGs, while the other formats save the decompressed pixel data.

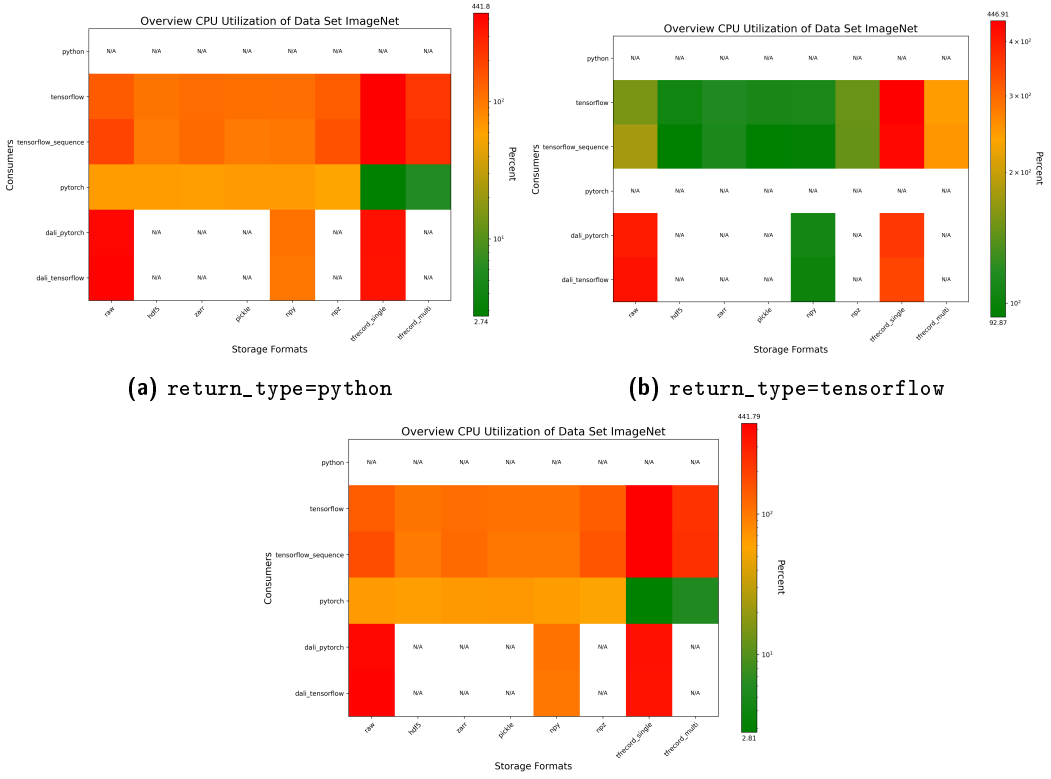
`npz` does compress the data, but not as much as JPEG as the results in Fig. 4.9 illustrate. Both TFRecord formats profit from the library having a function `tf.io.encode_jpeg` [74] and thus also take advantage of the JPEG compression.

4.3.2 Runtime

When comparing Fig. 4.10b to Figs. 4.10a and 4.10c, we see that the storage format has a non-negligible influence on the TensorFlow and sequential TensorFlow consumers. Most runs with `return_type=tensorflow` are slower with these consumers than for the other two return types.

Comparing the results of the runs using the different conventional consumers in Fig. 4.10, we observe that the PyTorch consumer is the fastest option for all storage formats. Adding the DALI consumers to the comparison, they achieve the fastest overall results when paired with the `raw` or `tfrecord_single` storage format.

The storage formats `tfrecord_single` and `tfrecord_multi` yield the slowest results by far for the conventional data loaders. Contrary to that, the DALI consumers are fastest when using `tfrecord_single`, followed by `raw`. In both cases, they are faster than any conventional consumer. In fact, the combination of the storage format `tfrecord_single` combined with the PyTorch DALI loader makes the fastest



(a) `return_type=python` **(b)** `return_type=tensorflow`

(c) `return_type=pytorch`

Figure 4.11: CPU utilization results of ILSVRC 2012

overall run of the suite. For `npy` the DALI consumers are fast too, but slower than the PyTorch consumer.

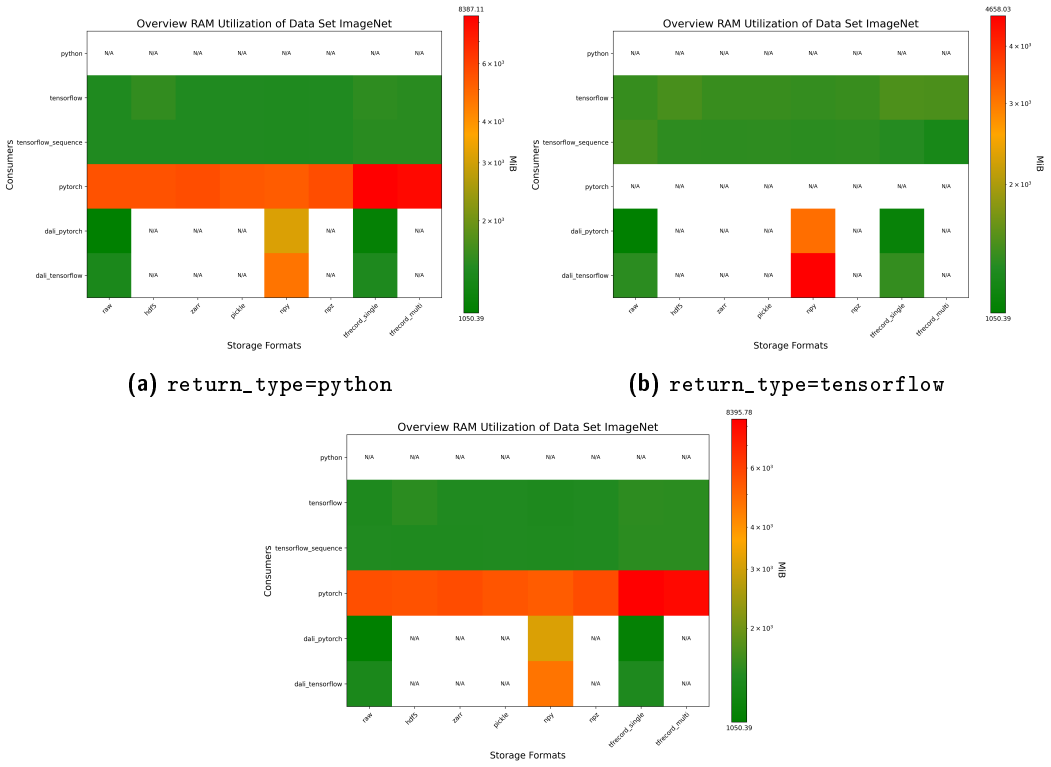
4.3.3 CPU Utilization

When comparing the subfigures in Fig. 4.11 the return type does not seem to have a large influence on the average CPU utilization for the ingestion of the ILSVRC 2012 data set. Most runs illustrated in Fig. 4.11b seem to require significantly less CPU compute compared to those in Figs. 4.11a and 4.11c, but only due to a shift in the color scale. The shift is caused by missing results of the PyTorch consumer, causing larger lower-end results.

In Fig. 4.11, we can nicely see a higher CPU utilization for the storage formats requiring JPEGs to be decoded. There is however one exception, the PyTorch consumer. It has a very low average CPU utilization in combination with the TFRecord file formats. This might be due to the extended runtime compared to the other storage formats. This may lower the average CPU utilization while PyTorch waits for the data to be read from disk.

For the other storage formats not storing JPEGs, the CPU utilization seems to be comparable when regarding a given consumer. In general, the PyTorch consumer has the lowest average CPU utilization.

The CPU utilization of the DALI consumers is comparable with that of the Tensor-



(a) `return_type=python` (b) `return_type=tensorflow`

(c) `return_type=pytorch`

Figure 4.12: RAM utilization results of ILSVRC 2012

Flow and sequential TensorFlow consumers, with the exception of the runs with the storage format `raw`. There we can observe a larger average CPU utilization. We cannot observe the same pattern for `tfrecord_single` runs, even though they also require the decoding of JPEGs.

4.3.4 RAM Utilization

Comparing the different subfigures of Fig. 4.12, it becomes clear that the RAM usage does not seem to depend on the return type. The results are comparable for runs sharing the same consumer and storage format.

The PyTorch consumer and the DALI consumers combined with the `npy` pipeline use the most RAM. The PyTorch consumer uses about three to five times the amount of RAM the other conventional consumers use. The RAM usage of the PyTorch consumer is largest when paired with one of the TFRecord storage formats.

For the storage format `npy`, the DALI PyTorch consumer uses roughly double the RAM compared to the conventional TensorFlow-based consumers. The DALI TensorFlow consumer uses about three times that amount. The RAM consumed by the DALI consumers when paired with the other storage formats is more moderate.

Other than the PyTorch consumer paired with the TFRecord storage formats and the DALI consumers paired with the storage format `npy`, the storage format does not seem to have a significant role in the RAM usage of a given consumer.

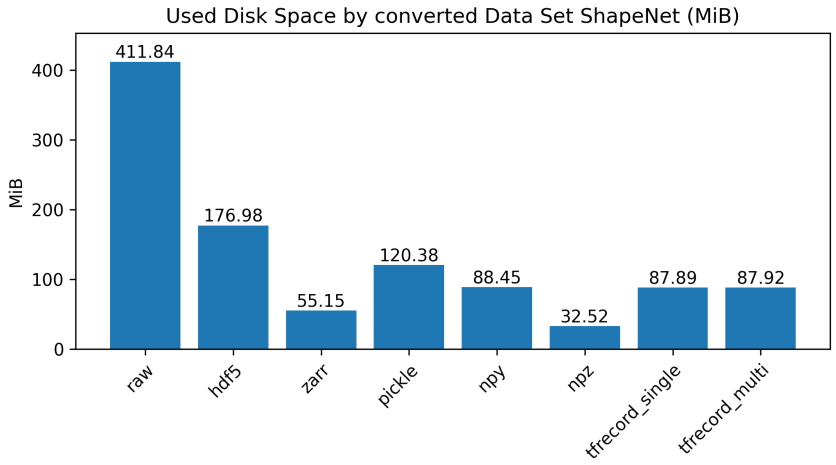


Figure 4.13: Disk space used by conversions of ShapeNet data set

4.4 ShapeNet

For the benchmark, models from the two randomly chosen categories `tower` and `knife` were used. This results in a total of 557 models. The chosen batch size was 16 and the number of threads used was 12 (configuration file A.4).

Due to technical difficulties, results for runs combining the TensorFlow or sequential TensorFlow consumers paired with the storage formats `tfrecord_single` or `tfrecord_multi` could not be executed on the GPU node. They were omitted.

4.4.1 Used Disk Space

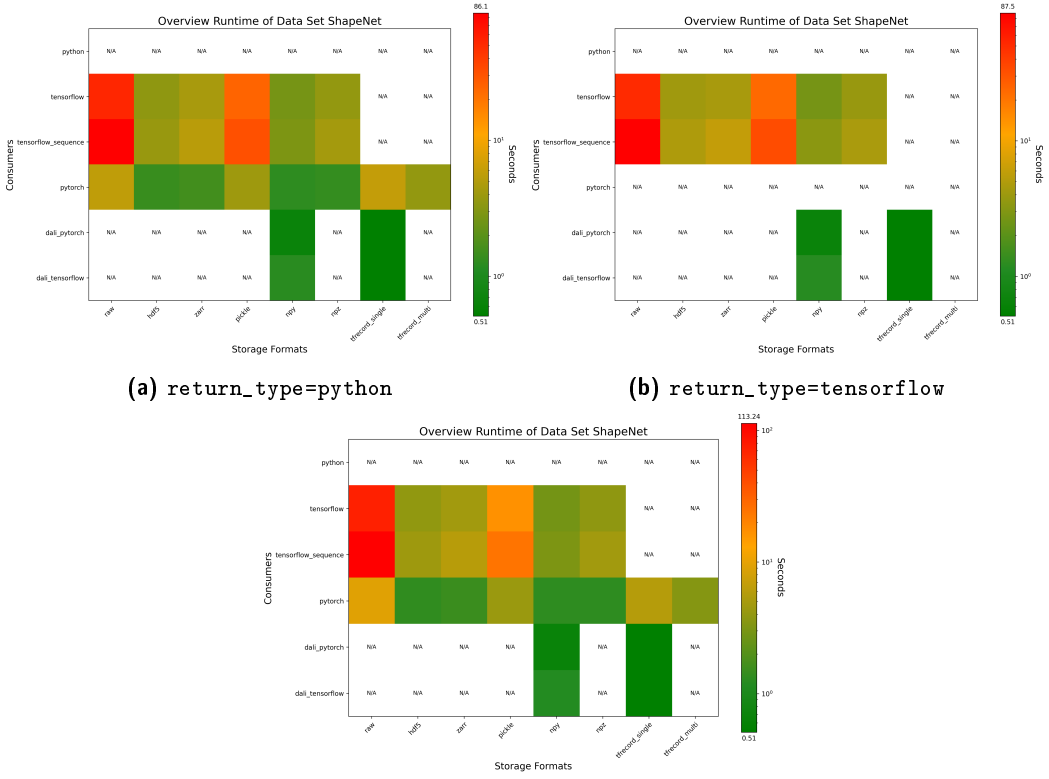
As can be seen in Fig. 4.13, the original storage format uses the most disk space. It is 225% larger compared to the next largest format. This can be explained by the original models partly containing images for the textures, as well as two file types being provided for every model. The models are present as OBJ [75] as well as Binvox files [76]. The models being present in different formats adds redundancy, which is not present in the conversions. They only save the faces and vertices of the models.

The `npz` format compresses the data the most. As can be seen in Fig. 4.13 it uses the least amount of memory on disk, being efficient at compressing the lists of faces and vertices.

4.4.2 Runtime

The subfigures in Fig. 4.14 illustrate, that there is a significant variance in runtime between the different return types. Although the absolute runtime changes, the relative ordering between the different runs mostly stays the same.

Comparing Figs. 4.14a and 4.14b to Fig. 4.14c, the runs with the storage format `raw` and the TensorFlow consumer are significantly faster (∼30%) when using the return type `python` or `tensorflow`, compared to `pytorch`. The storage format `pickle` read by the TensorFlow consumer shows a similar speed up when using the return type



(a) `return_type=python` (b) `return_type=tensorflow`

(c) `return_type=pytorch`

Figure 4.14: Runtime results of ShapeNet

`pytorch` rather than `python` or `tensorflow`. The latter observation also holds for the sequential TensorFlow consumer.

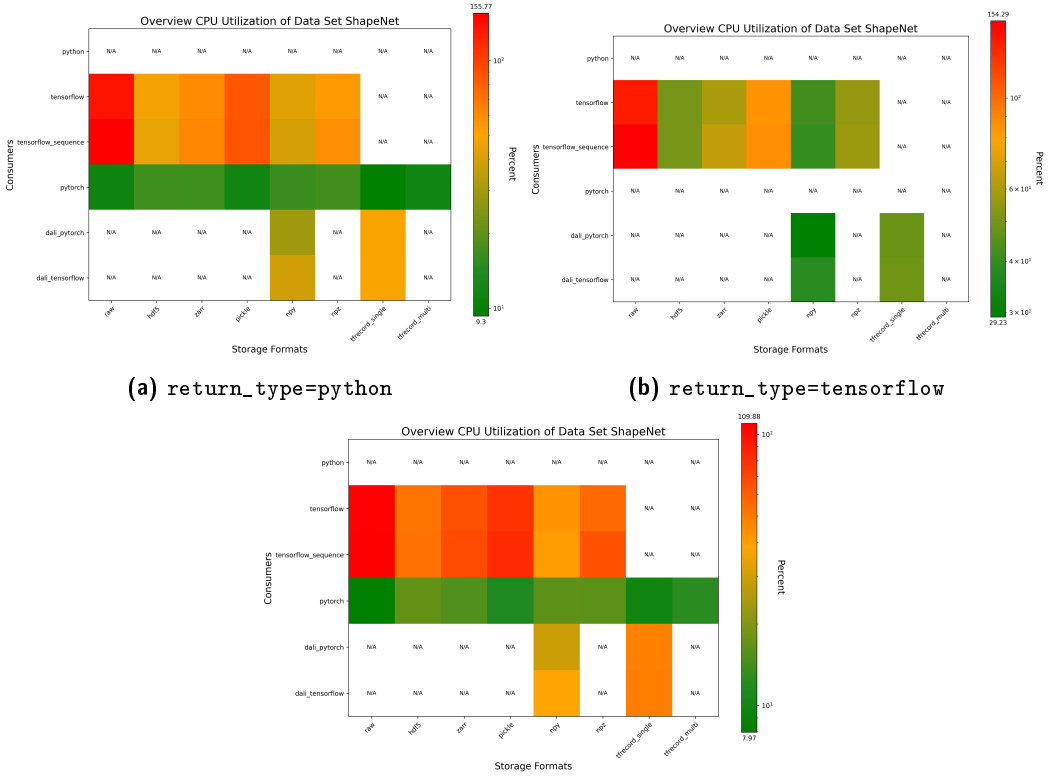
Regarding the results of the runs using a conventional consumer and a fixed storage format, the PyTorch consumer is always the fastest, as can be seen in Fig. 4.14. For both storage formats with available DALI pipelines the PyTorch consumer is outperformed by the DALI consumers. They show the best results using the `tfrecord_single` storage format.

Loading the data from the original OBJ files is the slowest option for all consumers. The runs using the storage format `pickle` or one of the TFRecord file formats prove to have a long runtime as well when paired with the conventional consumers. The conventional consumers perform best using the `npy` storage format.

4.4.3 CPU Utilization

As we can see in Fig. 4.15a and 4.15c the PyTorch consumer has the smallest average CPU utilization. This leads to a shift in the color scale of Fig. 4.15b, where it is missing. This makes the TensorFlow and sequential TensorFlow consumers seem to have a significantly smaller CPU utilization for the return type `tensorflow` compared to the others. This is not the case and only appears like that due to the shifted color scale.

As can be seen in Fig. 4.15c the conventional consumers require less CPU compute



(a) `return_type=python` (b) `return_type=tensorflow`

(c) `return_type=pytorch`

Figure 4.15: CPU utilization results of ShapeNet

for the return type `pytorch` when using the `raw` storage format. For the other storage formats, there is no clear trend for the conventional consumers regarding the average CPU utilization. Runs across different return types are comparable regarding this metric.

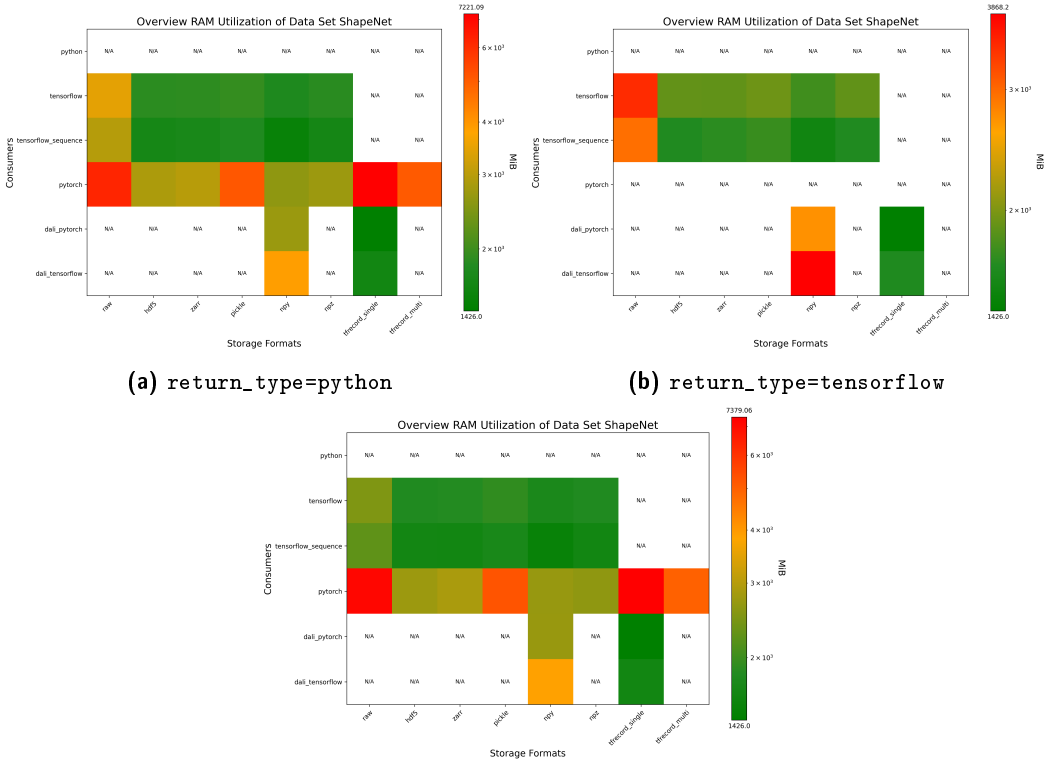
As illustrated in Fig. 4.15, the average CPU utilization of the DALI TensorFlow consumer is comparable with that of the TensorFlow or sequential TensorFlow consumers for the `npy` storage format. The CPU utilization of the DALI PyTorch consumer is about 25% lower. For the runs using the storage format `tfrecord_single` the DALI consumers have a nearly identical average CPU utilization.

4.4.4 RAM Utilization

In Fig. 4.16 we see that for a given storage format, the PyTorch consumer has the highest average RAM utilization. Only for the storage format `npy`, the DALI TensorFlow consumer shows a significantly higher average RAM consumption than the PyTorch consumer.

The RAM usage of the TensorFlow and sequential TensorFlow consumers paired with the `raw` storage format increases significantly when the return type is `tensorflow` or `python` (Figs. 4.16a and 4.16b). In these cases, the difference to the PyTorch consumer only lies at around 10-15%.

Other than the above-mentioned cases, the RAM usage is consistent across different



(a) `return_type=python` **(b)** `return_type=tensorflow`

(c) `return_type=pytorch`

Figure 4.16: RAM utilization results of ShapeNet

return types for the TensorFlow and sequential TensorFlow consumers. There are more differences for the PyTorch consumer. For the storage format `raw`, the RAM utilization increases by about 15% when using the return type `pytorch` instead of `python`. The other storage formats show a more consistent RAM usage for the PyTorch consumer across different return types.

The lowest average RAM consumption is achieved by the DALI consumers paired with the `tfrecord_single` pipeline. The results are still comparable to those of the TensorFlow and sequential TensorFlow consumers for the `npy` storage format. The DALI PyTorch consumer uses less RAM than the DALI TensorFlow consumer for both available pipelines.

4.5 General Observations

There are some general trends in the results which can be observed over all data sets. For one, it strongly depends on the data set if the return type has a significant influence on the metrics or not. In some cases, like for OADAT (Section 4.1.2) and ShapeNet (Section 4.4.2), it can have an influence on runtime. In others, like ShapeNet (see Section 4.4.3 and Section 4.4.4), it can also have a big influence on the average RAM and CPU utilization. For most data sets and metrics, the return type had a negligible influence.

When using the PyTorch consumer there seems to be a trade-off between the RAM

usage and the time needed to consume the data set. The PyTorch consumer is usually the fastest conventional consumer but also requires more RAM. The DALI consumers combined with the `npy` storage format also achieved fast results and showed an increased RAM usage.

The above trade-off only holds in the mentioned cases. We can see an exception for ShapeNet in Fig. 4.16a and 4.16b, where the TensorFlow and sequential TensorFlow consumers utilize a lot of RAM whilst consuming the ShapeNet data set in the `raw` storage format and yet having a long runtime.

For most data sets, the consumers generally have a larger influence on the runtime than the storage format. OADAT does not quite follow this trend. As can be seen in Fig. 4.2 we generally have more variation in the runtimes of the different benchmark runs. This variation is also seen across different storage formats for a fixed consumer.

The TFRecord formats generally slow down the conventional consumers compared to the other storage formats. For this random access use case, the `tfrecord_multi` file format was faster than the `tfrecord_single` format in all runs where they share the same consumer.

Contrary to the conventional consumers, the `tfrecord_single` format performed well in combination with the DALI consumers. We can make use of the DALI-TFRecord combination in the case of ILSVRC 2012, where a fast runtime (Fig. 4.10) with a low RAM consumption (Fig. 4.12) could be achieved (Section 4.3.1), but at the cost of a relatively high average CPU utilization (Fig. 4.11). As seen in Fig. 4.9, this storage format also uses very little space on disk compared to most other storage formats (Fig. 4.9).

The average CPU utilization follows a trend across the different benchmarked data sets. The benchmark runs including the PyTorch consumer generally seem to show a lower average CPU utilization than the other runs with the same storage format. Further, the runs including a TFRecord storage format, except for the ones combined with the PyTorch consumer, show a relatively high average CPU load. In the case of the conventional consumers, this can be explained through the fact, that to retrieve sample i of the data set, the first $i - 1$ samples first need to be skipped. This procedure is necessary to generate i.i.d. samples. For more on this, refer to Section 6.1.

Additionally, the data stored in the TFRecord files needs to be deserialized or decoded from JPEG, depending on the data set. This also requires additional CPU compute. The same phenomenon can also be observed for the storage formats `pickle` and `npz`, which employ serialization or compression. Their average CPU load is often slightly higher than that of the remaining storage formats. The difference is not as large as for the TFRecord file formats.

Depending on the data set, there can also be a lot of variance in the size that the converted data sets need on disk. In some cases, like ShapeNet, the size can be reduced drastically (Fig. 4.13). In other cases, like ILSVRC 2012, the needed size increases in nearly all cases (Fig. 4.9). For OADAT the size of the conversions are more or less similar to the original (Fig. 4.1).

4.6 Further Results

If there is interest in further results, they can be found in the directory `paper/paper_results` [77] of the project repository [5]. The subdirectories contain the raw results (Section 2.3.4) of the different suites. Corresponding graphs can be generated by executing `python paper/paper_results/create_graphs.py`. The graphs are generated in the subdirectory as they would be in the `result` directory when executing the benchmark (Section 2.3.4).

When using the script to create the graphs out of the results, the additional CLI option `--suites` is available. It enables the possibility to only generate graphs for the results in specified subdirectories of `paper/paper_results`. Then not all graphs need to be created, if not all are of interest.

The additional results include suites run on the same GPU node as the discussed results. They were generated with the command `python benchmarks/benchmarks.py --name_prefix gpu-node_`. Therefore, all preconfigured suites including the test data set have been run for all valid consumer, storage format, return type, and device combinations.

Similarly, results have been generated on a CPU node of SDSC using the command `python benchmarks/benchmarks.py --name_prefix cpu-node_`. The CPU node has 320 GB of RAM and two `Intel(R) Xeon(R) Platinum 8268`, each with 24 cores and 48 hardware threads, out of which 64 are available on the node.

The results generated on the CPU node include the missing runs for the ICON and ShapeNet data set, which needed to be omitted on the GPU node due to technical difficulties. The issues did not occur inn the CPU-only environment.

Chapter 5

Discussion

5.1 Benefits of Framework

As we have seen in Chapter 4 and summarized in Section 4.5, there can be substantial differences when it comes to the runtime, average CPU utilization, and average RAM consumption during DSI. Especially for larger DL projects it is very worthwhile to invest the time in the evaluation of different storage and loading options.

Thanks to the diverse metrics, data scientists can choose the best option based on their requirements and system limitations. If runtime is the only concern, the fastest combination of storage format, data loader, and return type can be chosen. If the CPU or RAM is shared or the space on disk is constrained, these factors can also be taken into account. As seen in Chapter 4, there may be large differences for these metrics too.

As described in Section 2.4, it is relatively easy to add a new data set, consumer, or storage format to the framework, if it does not contain the required options out of the box. Due to the significant performance differences across runs and data sets (Chapter 4) we are convinced, that it is worth the investment to add and benchmark custom implementations. If combinations of interest are already present in the framework, it is only a thing of a few minutes to create a configuration file and start the benchmark execution. Choosing a high-performing option will save expensive stalls of compute infrastructure and training time down the road.

5.2 General Recommendations

For very small projects, a potential user might be hesitant to invest the time to run the benchmarks on their system. In this case, the observations of Section 4.5 build a base for some general recommendations. They might not hold for every data set and every system, but can be used as a starting point.

5.2.1 Recommendations for fast Runtime

In case only runtime is relevant, PyTorch is probably the best framework option. Out of the conventional consumers, it is the fastest, but it has a high RAM usage.

In general, the most convenient return type option is recommended as its effects are usually negligible. OADAT and ShapeNet form exceptions, as the return type made a significant difference (Sections 4.1.2 and 4.4.2) in these cases.

For the storage formats `pickle`, `npy`, and `npz` usually performed best. The performance may vary between these different options. If the data set of interest was mentioned in this thesis, a look at the according section in Chapter 4 will help to choose the best option. Otherwise, the most convenient or compact option, usually `npz` or `pickle` based on our results, should be a good choice. Our recommendation is to only use Pickle if the data is pickled by the user themselves, due to security issues with Pickle [12].

An alternative to PyTorch would be to define a NVIDIA DALI pipeline if the samples are not required to be i.i.d. (see Section 6.2). The `tfrecord_single` pipeline performed best. However, this is of limited use if the samples are not images since we do not have access to the unserialized data. See Section 6.2 for an explanation. Alternatively, the `npy` pipeline usually also belonged to one of the fastest storage format options. If the samples are required to be i.i.d., refer to Section 7.4.

5.2.2 Recommendations for limited Compute Resources

If CPU or RAM resources are limited, the most friendly options based on our results (Chapter 4) were usually the TensorFlow or sequential TensorFlow consumers, combined with the storage formats `hdf5`, `zarr`, `pickle`, `npy`, or `npz`.

Using a DALI setup with the TFRecord reader [57] usually uses relatively little RAM, but can have an elevated CPU utilization. This option, if applicable to the concrete case (see Section 6.2 for details) is also relatively fast.

If only CPU usage is a concern, but RAM is not, the PyTorch data loader has shown an exceptionally low average CPU utilization across all data sets (Chapter 4). It is a fast option for most runs but has shown the highest RAM consumption in a majority of them.

5.2.3 Recommendations for limited Disk Space

As we have seen in the different graphs, the impact of the storage format on the used disk space varies a lot depending on the data set. Therefore, there are no general recommendations here. For details, see Section 4.5.

Chapter 6

Limitations

Unfortunately, there are some limitations to DSI-Bench. They are described in this chapter for completeness. If there are fixes, they are mentioned in Chapter 7.

6.1 Limitations of TFRecords

There proved to be various issues with the TFRecord storage formats. For one, there were technical difficulties in CUDA environments for the data sets ShapeNet and ICON, as mentioned at the start of Sections 4.2 and 4.4. With runs using the TensorFlow and sequential TensorFlow consumers as well as the storage formats `tfrecord_single` or `tfrecord_multi`, there were unexplainable errors of the type `OutOfRangeError: End of sequence`, although the data set was not fully consumed. These errors appeared at random points during the consumption. These combinations ran fine in CPU-only environments and the according results can be found in the DSI-Bench repository [5] under `paper/paper_results` [77].

Another limitation of TFRecords is that it is designed for fast sequential access [43]. This makes random access, which is necessary for i.i.d. sampling, relatively slow. To access the i^{th} sample, the first $i - 1$ samples of the data set need to be skipped. When consuming the data set in a random order, this slows down the data ingestion when using the conventional data loaders, as also observed in Section 4.5.

6.2 Limitations of DALI

The NVIDIA DALI library is still relatively new (released in 2018 [17]) and does not offer readers for a lot of different formats yet. Thus, in most cases, pipeline definitions could only be created for two storage formats (three for ILSVRC 2012).

For the `tfrecord_single` pipeline, there is another large limitation. Due to the higher dimensional features of most benchmarked data sets, the features need to be serialized before being able to save them to TFRecord files. The DALI pipeline for `tfrecord_single` is currently not able to deserialize these tensors. Hence, the runtimes are fast, but the results are unusable in practice without having access to the deserialized data. There is one exception to this limitation. For the ILSVRC 2012 data set, the `tfrecord_single` pipeline is able to decode the images, as the

data is not serialized conventionally but encoded as JPEGs. See Section 7.4 for more.

The readers in the DALI pipelines also have the attribute `random_shuffle` [44, 56, 57] set to `True`. This causes the data to be read sequentially in batches of size `initial_fill` (1024 by default [44, 56, 57]) and be randomly sampled from there [44, 56, 57]. Therefore, the samples are not i.i.d., which is a requirement for a lot of DL applications. See Section 7.4 for more.

6.3 Data Set Sizes

Most benchmarked subsets of the used data sets only had a size of a few 10 MB to a few 100 MB. Due to time constraints, larger subsets could not be benchmarked. It might have been the case in some scenarios that the whole data was loaded into memory. This is not the case in general DL applications and might have an influence on the results. See Section 7.5 for more.

6.4 Optimizations

The benchmarking framework consists of various different components, as described in Section 2.2. Making all these components work together for the various implemented data sets, consumers, and storage formats listed in Section 2.3, required a lot of effort for cross compatibility. These efforts e.g. include that the different storage format-specific data set classes all return the same type of object, while it may have been more efficient to directly return another type or structure for a specific format.

These efforts in the name of compatibility entailed some workarounds and missed opportunities for optimizations. Some of these possibilities are listed in Section 7.3. Nevertheless, we believe that although the absolute benchmark results are slightly worse than necessary, they still provide a good relative orientation between the different storage formats and consumers for the various data sets.

Another limitation of the benchmark is, that there is currently no way of marking the `raw` storage format as equal to one of the others. In the case of OADAT and ICON, this causes unnecessary benchmark runs and could easily be optimized. Alternatively, the `raw` storage format could also easily be omitted using the CLI option `converter_names`.

6.5 Lacking Preprocessing

Preprocessing is often an important part of modern DSI pipelines. This is not accounted for in the benchmark, as it was out of scope. Currently, if any sort of preprocessing is necessary, it is done before converting the data sets, like for ILSVRC 2012 (Section 2.3.3). In this case, the images are resized before saving them. However, certain preprocessing steps, like random cropping, rotation, or resizing, cannot be done in advance. These are currently not pursued or measured in any way.

How one could add preprocessing to the framework is outlined in Section 7.2. Depending on how the preprocessing steps of the DSI pipelines are implemented, the resulting runtime and compute utilization could vary drastically. Representing this important aspect of the DSI process in DSI-Bench would definitely enrich the framework and result in valuable insights.

Chapter 7

Further Work

There are further options to improve DSI-Bench. In this chapter some of these are discussed to inspire members of the data science community to contribute to this project, trying to move efficient DSI more into focus.

7.1 Additional Features

Further data sets, storage formats, and consumers could be added. Having numerous options being usable out of the box makes it easier for data scientists to benchmark relevant combinations of data sets, storage formats, and data loaders on their systems. Depending on the system on which the benchmark is run, the results may differ due to various possible bottlenecks, such as CPU, memory, or disk speed.

Possibilities for data sets include a text or audio-based data set, as well as graph data sets, which are not yet represented in the benchmark. Possible examples include the IMDb Movie Reviews dataset [78], AudioMNIST [79], or the Reddit graph data set [80]. Further storage formats could include TIFF [81] or NetCDF4 [82]. Additional consumer options could include a consumer using JAX [83] or a JAX-based NVIDIA DALI consumer [84].

When extending the benchmark, it would be convenient to do so without needing to edit the benchmark script. This would make the framework even easier to adapt to specific use cases if none of the currently available options satisfy the needs of the user. Ideally, the user would only need to create the implementation and be able to use it.

7.2 Additional Benchmark Stages

Another useful addition would be some form of a second benchmark stage after loading the data to implement and measure preprocessing. Preprocessing is an important component of most DSI pipelines and it would thus be interesting to see how different preprocessing implementations further influence the runtime. The performance of in-GPU preprocessing compared to conventional preprocessing by the CPU would be especially interesting. Such options are made possible by

various NVIDIA DALI functions [26], which can be used in the pipeline definitions. However, it might make sense to add this step as a separate stage and illustrate the results separately from the loading, as the concrete preprocessing performed is very use-case-specific.

Contrary to a second stage, it would be useful to add a pre-stage, where the data set and loader objects are created. The conversion of the data sets, as well as the creation of the data loaders, are not included in the benchmark, only the consumption. If there is a significant speed-up due to prefetching or similar, this in return probably comes at a cost during data set creation, as there is no such thing as a free lunch. Therefore, the inclusion of the data loader and data set creation in an additional pre-stage could provide useful insights about such trade-offs.

7.3 Optimizations

It would be interesting to explore the effects of various optimizations further. To make the different components of the benchmark cross compatible, they were held very generic, as mentioned in Section 6.4. Analyzing further optimization possibilities would result in interesting results. Optimizations, such as caching, prefetching, specific preprocessing implementations, or more low-level options, like setting the CPU affinity, could very well be worthwhile when processing large data sets during extended periods of training.

An additional pre-stage or second stage, as described previously in Section 7.2, would be useful to see if there are other trade-offs being made when the consumption of the data set is optimized.

7.4 Data Loading using NVIDIA DALI

As mentioned in Section 6.2, the DALI pipelines for the storage format `tfrecord_single` were included in the benchmark, however are of limited practical use for the used data sets. It would be interesting to see how the runtimes evolve when additional deserializing of the data takes place and if this combination would still be the fastest option to load the data. A possible solution would be to make use of `tf.io.encode_jpeg` [74] instead of serializing the samples. However, since JPEG is a lossy compression format [85], there would be some loss in accuracy of the data. Practical experiments would be required to see if the loss in precision has any effects on DL applications, or if it is worth the possible speed-up of the DSI process.

Another addition would be to benchmark a version of the DALI pipelines, which return i.i.d. samples. As mentioned in Section 6.2, the returned samples are currently not i.i.d. If instead of the attribute `random_shuffle shuffle_after_epoch` [44, 56, 57] is set to `True`, the returned sample should be i.i.d. Since the first shuffle is done during loader construction, the first epoch would already contain shuffled samples.

For use cases that require the samples of the data set to be i.i.d. these results would be very interesting. Since the first shuffling happens during loader construction,

a pre-stage as mentioned in Section 7.2 would be necessary. The performance might be comparable when consuming the data set, but the first shuffle during construction might induce a significant overhead, especially for large data sets.

In case of adding preprocessing to the benchmark, it makes sense to slightly restructure the benchmark to not limit the number of pipelines per data set and storage format to one. Being able to have multiple pipelines for a single storage format makes it easier to compare different preprocessing implementations with each other.

7.5 Miscellaneous further Work

Generating a useful PDF or HTML report to summarize the benchmarks would add convenience for the user. If we have a lot of combinations for consumers, storage formats, and several different data sets, a large number of graphs are generated. To provide a better overview and make the results easier to analyze, a more comprehensive report may prove to be helpful.

Further work includes resolving the technical difficulties in CUDA environments with the ShapeNet and ICON data sets. As mentioned at the start of Sections 4.2 and 4.4 and in Section 6.1, the runs using a TFRecord file format and either the TensorFlow or sequential TensorFlow consumer needed to be omitted.

As described in Section 6.3, the sizes of the benchmarked data sets were limited. This might influence the results compared to large DL projects, where data sets can have several TB. Therefore, results with larger subsets that do not fit into main memory would also contain valuable information regarding DSI performance.

Bibliography

[1] C. Sun, A. Shrivastava, S. Singh, and A. Gupta, "Revisiting unreasonable effectiveness of data in deep learning era," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct. 2017.

[2] S. Ambatipudi and S. Byna, "A comparison of hdf5, zarr, and netcdf4 in performing common i/o operations," Jul. 2022. DOI: 10.48550/arXiv.2207.09503.

[3] M. Zhao *et al.*, "Understanding data storage and ingestion for large-scale deep recommendation model training: Industrial product," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22, New York, NY, USA: Association for Computing Machinery, 2022, 1042–1057, ISBN: 9781450386104. DOI: 10.1145/3470496.3533044.

[4] I. Ofeidis, D. Kiedanski, and L. Tassiulas, *An overview of the data-loader landscape: Comparative performance analysis*, 2022. DOI: https://doi.org/10.48550/arXiv.2209.13705. arXiv: 2209.13705 [cs.DC].

[5] J. Hochstrasser, F. Ozdemir, and M. Meyer, *Dsi-bench repository*, Aug. 2024. [Online]. Available: https://github.com/Jan-Hoc/DSI-Bench (visited on 08/26/2024).

[6] F. Ozdemir, B. Lafci, X. L. Deán-Ben, D. Razansky, and F. Perez-Cruz, *Oadat: Experimental and synthetic clinical optoacoustic data for standardized image processing*, 2023. arXiv: 2206.08612 [eess.IV]. [Online]. Available: https://arxiv.org/abs/2206.08612.

[7] I. partnership (DWD, MPI-M, DKRZ, KIT, and C2SM), *Icon release 2024.01*, 2024. DOI: 10.35089/WDCC/IconRelease01. [Online]. Available: https://doi.org/10.35089/WDCC/IconRelease01.

[8] O. Russakovsky *et al.*, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015. DOI: 10.1007/s11263-015-0816-y.

[9] A. X. Chang *et al.*, "ShapeNet: An Information-Rich 3D Model Repository," Stanford University — Princeton University — Toyota Technological Institute at Chicago, Tech. Rep. arXiv:1512.03012 [cs.GR], 2015.

[10] "Hdf5 high-performance data management and storage suite," HDF Group. (), [Online]. Available: https://www.hdfgroup.org/solutions/hdf5/ (visited on 08/16/2024).

[11] A. Miles *et al.*, *Zarr-developers/zarr-python: V2.4.0*, version v2.4.0, Apr. 2020. DOI: 10.5281/zenodo.3773450. [Online]. Available: https://doi.org/10.5281/zenodo.3773450 (visited on 08/17/2024).

[12] *Pickle documentation*, Python Software Foundation. [Online]. Available: https://docs.python.org/3.10/library/pickle.html (visited on 08/16/2024).

[13] *Numpy `numpy.lib.format` documentation*, NumPy Developers. [Online]. Available: https://numpy.org/doc/stable/reference/generated/numpy.lib.format.html (visited on 08/16/2024).

[14] *Tfrecord and `tf.train.Example`*, TensorFlow. [Online]. Available: https://www.tensorflow.org/tutorials/load_data/tfrecord (visited on 08/16/2024).

[15] M. Abadi *et al.*, *TensorFlow, Large-scale machine learning on heterogeneous systems*, Nov. 2015. DOI: 10.5281/zenodo.4724125.

[16] J. Ansel *et al.*, "PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation," in *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*, ACM, Apr. 2024. DOI: 10.1145/3620665.3640366. [Online]. Available: https://pytorch.org/assets/pytorch2-2.pdf.

[17] *Nvidia data loading library (dali) documentation*, NVIDIA. [Online]. Available: https://docs.nvidia.com/deeplearning/dali/user-guide/docs/index.html (visited on 07/14/2024).

[18] D. Amodei, D. Hernandez, G. Sastry, J. Clark, G. Brockman, and I. Sutskever. "Ai and compute." English, OpenAI. (May 2018), [Online]. Available: https://openai.com/index/ai-and-compute/ (visited on 07/11/2024).

[19] M. Andersch *et al.*, "Nvidia h100 tensor core gpuarchitecture," Mar. 2022. [Online]. Available: https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper?ncid=no-ncid (visited on 07/11/2024).

[20] J. Mohan, A. Phanishayee, A. Raniwala, and V. Chidambaram, "Analyzing and mitigating data stalls in dnn training," 2021. arXiv: 2007.06775 [cs.DC]. [Online]. Available: https://arxiv.org/abs/2007.06775 (visited on 08/21/2024).

[21] I. Sarker, "Deep learning: A comprehensive overview on techniques, taxonomy, applications and research directions," *SN Computer Science*, vol. 2, no. 6, p. 420, Aug. 18, 2021. DOI: 10.1007/s42979-021-00815-1.

[22] T. Moreau *et al.*, "Benchopt: Reproducible, efficient and collaborative optimization benchmarks," in *NeurIPS*, 2022. [Online]. Available: https://arxiv.org/abs/2206.13424.

[23] C. Lee, M. Yang, and R. A. Aydt, "Netcdf-4 performance report," 2008. [Online]. Available: https://api.semanticscholar.org/CorpusID:15588867.

[24] *Nvidia dali pipeline documentation*, NVIDIA. [Online]. Available: https://docs.nvidia.com/deeplearning/dali/user-guide/docs/pipeline.html (visited on 07/14/2024).

[25] *Nvidia dali `nvidia.dali.fn.readers` documentation*, NVIDIA. [Online]. Available: https://docs.nvidia.com/deeplearning/dali/user-guide/docs/operations/nvidia.dali.fn.readers.html#nvidia-dali-fn-readers (visited on 07/14/2024).

[26] *Nvidia dali operation reference*, NVIDIA. [Online]. Available: https://docs.nvidia.com/deeplearning/dali/user-guide/docs/supported_ops.html (visited on 08/20/2024).

[27] "Zarr homepage," Zarr development team. (), [Online]. Available: https://zarr.dev (visited on 08/16/2024).

[28] *Nvidia dali pytorch plugin reference*, NVIDIA. [Online]. Available: https://docs.nvidia.com/deeplearning/dali/user-guide/docs/plugins/pytorch_tutorials.html (visited on 08/17/2024).

[29] *Nvidia dali tensorflow plugin reference*, NVIDIA. [Online]. Available: https://docs.nvidia.com/deeplearning/dali/user-guide/docs/plugins/tensorflow_tutorials.html (visited on 08/17/2024).

[30] *Python `None` documentation*, Python Software Foundation. [Online]. Available: https://docs.python.org/3.10/library/constants.html?#None (visited on 08/24/2024).

[31] *Python `typing.Callable` documentation*, Python Software Foundation. [Online]. Available: https://docs.python.org/3.10/library/typing.html?#typing.Callable (visited on 08/24/2024).

[32] *Pytorch `torch.utils.data.DataLoader` documentation*, The Linux Foundation. [Online]. Available: https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader (visited on 07/14/2024).

[33] *Pytorch `torch.Tensor.to` documentation*, The Linux Foundation. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.Tensor.to.html#torch.Tensor.to (visited on 08/17/2024).

[34] *Tensorflow `tf.data.Dataset.padded_batch` documentation*, TensorFlow. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/data/Dataset#padded_batch (visited on 07/14/2024).

[35] *Tensorflow `tf.TensorSpec` documentation*, TensorFlow. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/TensorSpec (visited on 07/14/2024).

[36] *Tensorflow `tf.data.Dataset.from_generator` documentation*, TensorFlow. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/data/Dataset#from_generator (visited on 07/14/2024).

[37] J. Hochstrasser, F. Ozdemir, and M. Meyer, *`SysLoadBench`*, Benchmarking framework for Python to measure the system load and execution time of functions. [Online]. Available: https://github.com/Jan-Hoc/PySysLoadBench (visited on 08/16/2024).

[38] A. Collette. "`h5py` homepage." (), [Online]. Available: h5py.org (visited on 07/17/2024).

[39] *Pickle `pickle.HIGHEST_PROTOCOL` documentation*, Python Software Foundation. [Online]. Available: https://docs.python.org/3/library/pickle.html#pickle.HIGHEST_PROTOCOLhttps://docs.python.org/3/library/pickle.html#pickle.HIGHEST_PROTOCOL (visited on 08/17/2024).

[40] *Numpy `numpy.save` documentation*, NumPy Developers. [Online]. Available: https://numpy.org/doc/stable/reference/generated/numpy.save.html (visited on 08/17/2024).

[41] *Numpy `numpy.load` documentation*, NumPy Developers. [Online]. Available: https://numpy.org/doc/stable/reference/generated/numpy.load.html (visited on 08/17/2024).

[42] *Numpy `numpy.savez_compressed` documentation*, NumPy Developers. [Online]. Available: https://numpy.org/doc/stable/reference/generated/numpy.savez_compressed.html (visited on 08/17/2024).

[43] *Tfrecords format details*, TensorFlow. [Online]. Available: https://www.tensorflow.org/tutorials/load_data/tfrecord#tfrecords_format_details (visited on 08/20/2024).

[44] *Nvidia dali `nvidia.dali.fn.readers.file` documentation*, NVIDIA. [Online]. Available: https://docs.nvidia.com/deeplearning/dali/user-guide/docs/operations/nvidia.dali.fn.readers.file.html#nvidia.dali.fn.readers.file (visited on 08/17/2024).

[45] *Nvidia dali `nvidia.dali.fn.decoders.image` documentation*, NVIDIA. [Online]. Available: https://docs.nvidia.com/deeplearning/dali/user-guide/docs/operations/nvidia.dali.fn.decoders.image.html (visited on 08/24/2024).

[46] *`concurrent.futures.ThreadPoolExecutor` documentation*, Python Software Foundation. [Online]. Available: https://docs.python.org/3.10/library/concurrent.futures.html#threadpoolexecutor (visited on 08/17/2024).

[47] *`concurrent.futures.ProcessPoolExecutor` documentation*, Python Software Foundation. [Online]. Available: https://docs.python.org/3.10/library/concurrent.futures.html#processpoolexecutor (visited on 08/17/2024).

[48] B. Slatkin, *Effective Python: 90 Specific Ways to Write Better Python* (Effective software development series), 2nd ed. Addison-Wesley, 2020, ISBN: 9780134853987. [Online]. Available: https://books.google.ch/books?id=QgIMvgEACAAJ.

[49] *Tensorflow `tf.data` documentation*, TensorFlow. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/data (visited on 08/24/2024).

[50] *Tensorflow `tf.data.Dataset` documentation*, TensorFlow. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/data/Dataset (visited on 08/17/2024).

[51] *Tensorflow `tf.keras.utils.Sequence` documentation*, TensorFlow. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/utils/Sequence (visited on 08/17/2024).

[52] *Tensorflow `tf.keras` module documentation*, TensorFlow. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras (visited on 08/17/2024).

[53] *Tensorflow `tensorflow.python.keras.utils.all_utils.OrderedEnqueuer` source code*, TensorFlow. [Online]. Available: https://github.com/tensorflow/tensorflow/blob/5bc9d26649cca274750ad3625bd93422617eed4b/tensorflow/python/keras/utils/data_utils.py#L675 (visited on 08/17/2024).

[54] *Nvidia dali `nvidia.dali.plugin.pytorch.DALIGenericIterator` documentation*, NVIDIA. [Online]. Available: https://docs.nvidia.com/deeplearning/dali/user-guide/docs/plugins/pytorch_plugin_api.html#nvidia.dali.plugin.pytorch.DALIGenericIterator (visited on 08/17/2024).

[55] *Nvidia dali `nvidia.dali.plugin.tf.DALIIterator` documentation*, NVIDIA. [Online]. Available: https://docs.nvidia.com/deeplearning/dali/user-guide/docs/plugins/tensorflow_plugin_api.html#nvidia.dali.plugin.tf.DALIIterator (visited on 08/17/2024).

[56] *Nvidia dali `nvidia.dali.fn.readers.numpy` documentation*, NVIDIA. [Online]. Available: https://docs.nvidia.com/deeplearning/dali/user-guide/docs/operations/nvidia.dali.fn.readers.numpy.html#nvidia.dali.fn.readers.numpy (visited on 08/17/2024).

[57] *Nvidia dali `nvidia.dali.fn.readers.tfrecord` documentation*, NVIDIA. [Online]. Available: https://docs.nvidia.com/deeplearning/dali/user-guide/docs/operations/nvidia.dali.fn.readers.tfrecord.html#nvidia.dali.fn.readers.tfrecord (visited on 08/17/2024).

[58] *Numpy `numpy.array` documentation*, NumPy Developers. [Online]. Available: https://numpy.org/doc/stable/reference/generated/numpy.array.html (visited on 08/24/2024).

[59] T. Khot, *Shapenet category mapping*. [Online]. Available: https://gist.githubusercontent.com/tejaskhot/15ae62827d6e43b91a4b0c5c850c168e/raw/5064af3603d509b79229f6931998d4e197575ad3/shapenet_synset_list (visited on 07/17/2024).

[60] J. Hochstrasser, F. Ozdemir, and M. Meyer, *Dsi-bench implementation test data set*, Aug. 2024. [Online]. Available: https://github.com/Jan-Hoc/DSI-Bench/blob/main/dsi_bench/datasets/test.py (visited on 08/26/2024).

[61] J. Hochstrasser, F. Ozdemir, and M. Meyer, *Dsi-bench `TensorflowSequenceConsumer` implementation*, Aug. 2024. [Online]. Available: https://github.com/Jan-Hoc/DSI-Bench/blob/main/dsi_bench/consumers.py#L266 (visited on 08/26/2024).

[62] J. Hochstrasser, F. Ozdemir, and M. Meyer, *Dsi-bench `DALIPytorchConsumer` implementation*, Aug. 2024. [Online]. Available: https://github.com/Jan-Hoc/DSI-Bench/blob/main/dsi_bench/consumers.py#L484 (visited on 08/26/2024).

[63] J. Hochstrasser, F. Ozdemir, and M. Meyer, *Dsi-bench `DALITensorflowConsumer` implementation*, Aug. 2024. [Online]. Available: https://github.com/Jan-Hoc/DSI-Bench/blob/main/dsi_bench/consumers.py#L553 (visited on 08/26/2024).

[64] conda contributors, *conda: A system-level, binary package and environment manager running on all major operating systems and platforms.* [Online]. Available: https://github.com/conda/conda (visited on 08/18/2024).

[65] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[66] S. Eustace and The Poetry contributors, *Poetry: Python packaging and dependency management made easy*. [Online]. Available: https://github.com/python-poetry/poetry (visited on 08/18/2024).

[67] J. Hochstrasser, F. Ozdemir, and M. Meyer, *Dsi-bench environment installation script*, Aug. 2024. [Online]. Available: https://github.com/Jan-Hoc/DSI-Bench/blob/main/environment/install-env.sh (visited on 08/26/2024).

[68] J. Hochstrasser, *Dsi-bench docker images*, Aug. 2024. [Online]. Available: https://hub.docker.com/r/majorhph/dsi-bench (visited on 08/26/2024).

[69] *Github actions documentation*, GitHub, Inc. [Online]. Available: https://docs.github.com/en/actions (visited on 08/18/2024).

[70] J. Hochstrasser, F. Ozdemir, and M. Meyer, *Dsi-bench: Docker section of readme.md*, Aug. 2024. [Online]. Available: https://github.com/Jan-Hoc/DSI-Bench/tree/main?tab=readme-ov-file#docker (visited on 08/26/2024).

[71] "Run:ai instance of sdsc." (), [Online]. Available: https://sdsc.run.ai/ (visited on 08/18/2024).

[72] J. Hochstrasser, F. Ozdemir, and M. Meyer, *Dsi-bench benchmark script*, Aug. 2024. [Online]. Available: https://github.com/Jan-Hoc/DSI-Bench/blob/main/benchmarks/benchmarks.py (visited on 08/26/2024).

[73] NVIDIA, *List of cuda compatible devices*. [Online]. Available: https://developer.nvidia.com/cuda-gpus (visited on 08/18/2024).

[74] *Tensorflow `tf.io.encode_lpeg` documentation*, TensorFlow. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/io/encode_jpeg (visited on 08/19/2024).

[75] Library of Congress, *Wavefront obj file format*. [Online]. Available: https://www.loc.gov/preservation/digital/formats/fdd/fdd000507.shtml (visited on 08/19/2024).

[76] P. Min, *Binvox*, http://www.patrickmin.com/binvox, 2004 - 2019. [Online]. Available: http://www.patrickmin.com/binvox (visited on 08/19/2024).

[77] J. Hochstrasser, F. Ozdemir, and M. Meyer, *Dsi-bench paper results*, Aug. 2024. [Online]. Available: https://github.com/Jan-Hoc/DSI-Bench/tree/main/paper/paper_results (visited on 08/26/2024).

[78] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, D. Lin, Y. Matsumoto, and R. Mihalcea, Eds., Portland, Oregon, USA: Association for Computational Linguistics, Jun. 2011, pp. 142–150. [Online]. Available: https://aclanthology.org/P11-1015.

[79] S. Becker, J. Vielhaben, M. Ackermann, K.-R. Müller, S. Lapuschkin, and W. Samek, "Audiomnist: Exploring explainable artificial intelligence for audio analysis on a simple benchmark," *Journal of the Franklin Institute*, 2023, ISSN: 0016-0032. DOI: https://doi.org/10.1016/j.jfranklin.2023.11.038. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0016003223007536.

[80] W. L. Hamilton, R. Ying, and J. Leskovec, *Inductive representation learning on large graphs*, 2018. arXiv: 1706.02216 [cs.SI]. [Online]. Available: https://arxiv.org/abs/1706.02216.

[81] *Tiff*, Aldus Corporation, Jun. 1992. [Online]. Available: https://www.itu.int/itudoc/itu-t/com16/tiff-fx/docs/tiff6.pdf (visited on 08/20/2024).

[82] Unidata, *Netcdf4 file format specification*, 2011. DOI: https://doi.org/10.5065/D6H70CW6. [Online]. Available: https://docs.unidata.ucar.edu/netcdf/index.html.

[83] J. Bradbury *et al.*, *JAX: Composable transformations of Python+NumPy programs*, version 0.3.13, 2018. [Online]. Available: http://github.com/google/jax.

[84] *Nvidia dali jax plugin reference*, NVIDIA. [Online]. Available: https://docs.nvidia.com/deeplearning/dali/user-guide/docs/plugins/jax_tutorials.html (visited on 08/20/2024).

[85] C. Lilley, *Jpeg: Still image compression standard*, JPEG is a lossy compression standard for still images., 1996. [Online]. Available: https://www.w3.org/Graphics/JPEG/ (visited on 08/20/2024).

Appendix A

Labeled Benchmark Result

A.1 Results OADAT Data Set

Configuration File A.1: OADAT Data Set

```
dataset_name: 'OADAT'
dataset_type: 'oadat'
path: '<dataset-path>'
filename: 'SWFD_semicircle_RawBP_1k_ss.h5'
key:
  - 'sc_BP'
  - 'sc,lv128_BP'
batch_size: 32
num_threads: 12
```

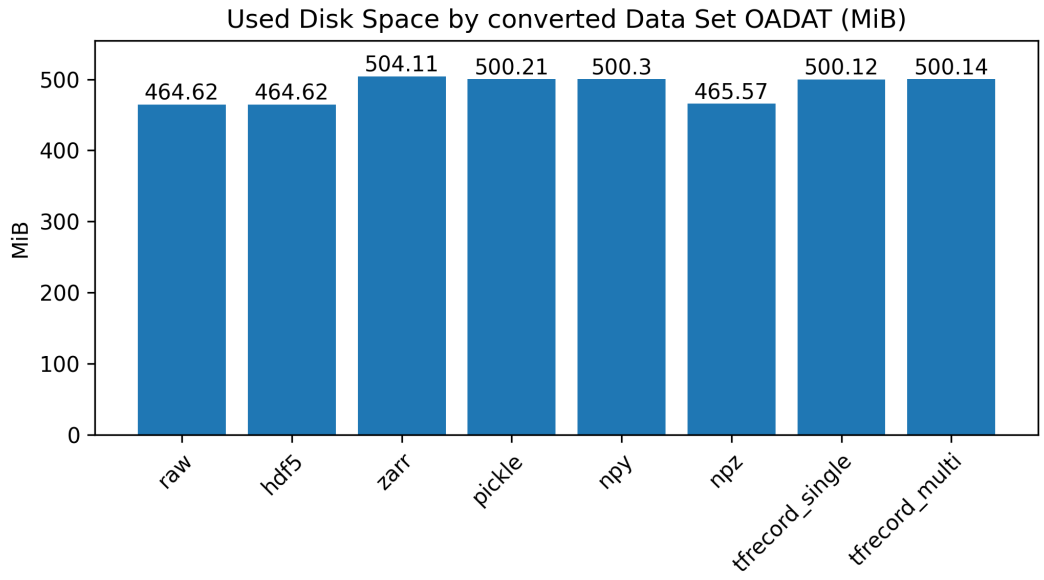
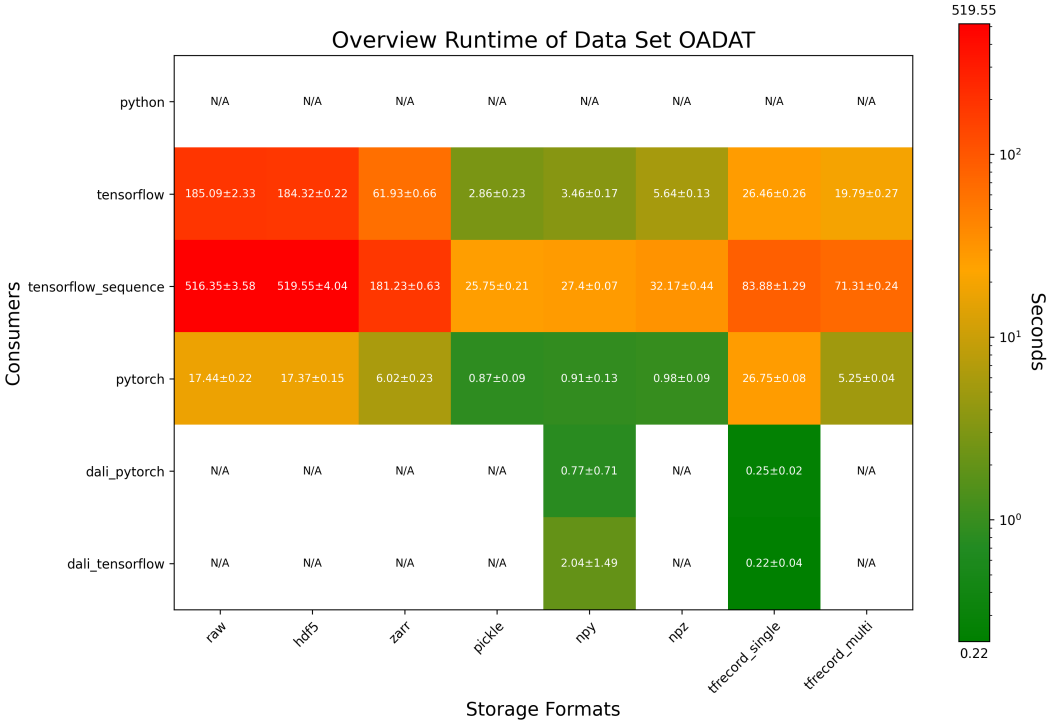
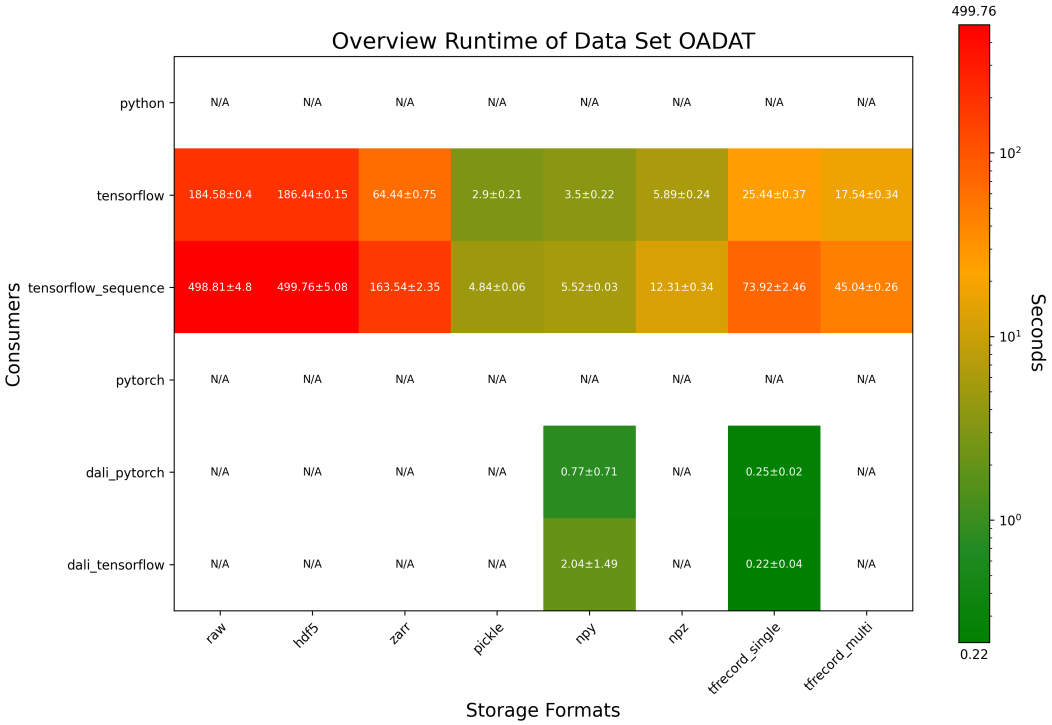


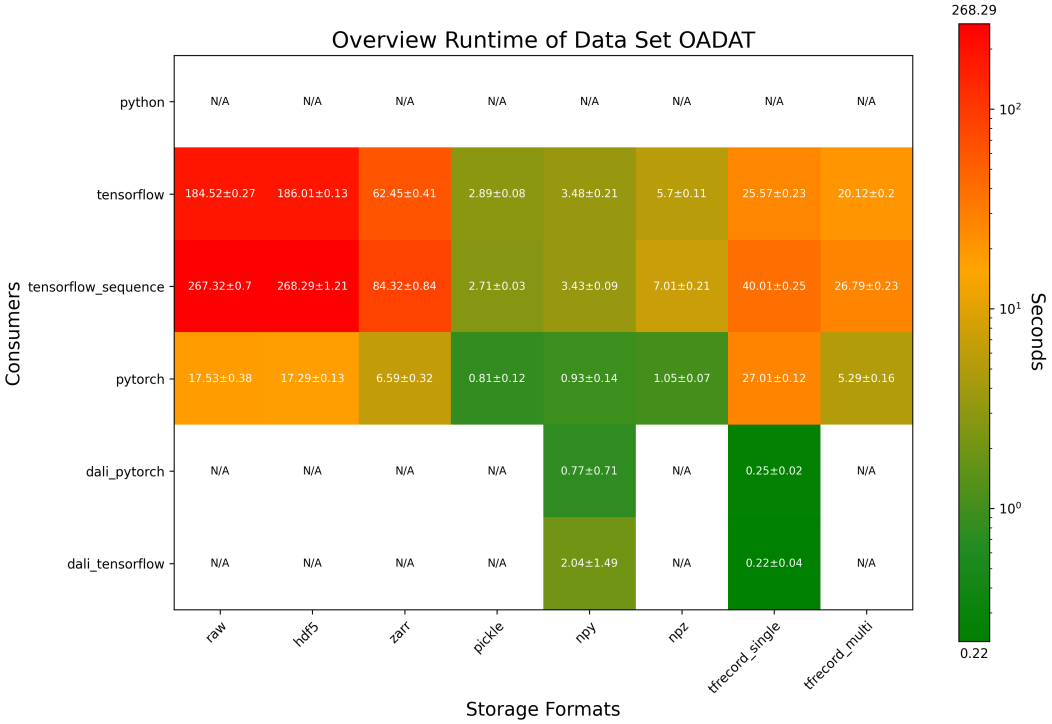
Figure A.1: Disk space used by conversions of OADAT data set



(a) `device=gpu, return_type=python`

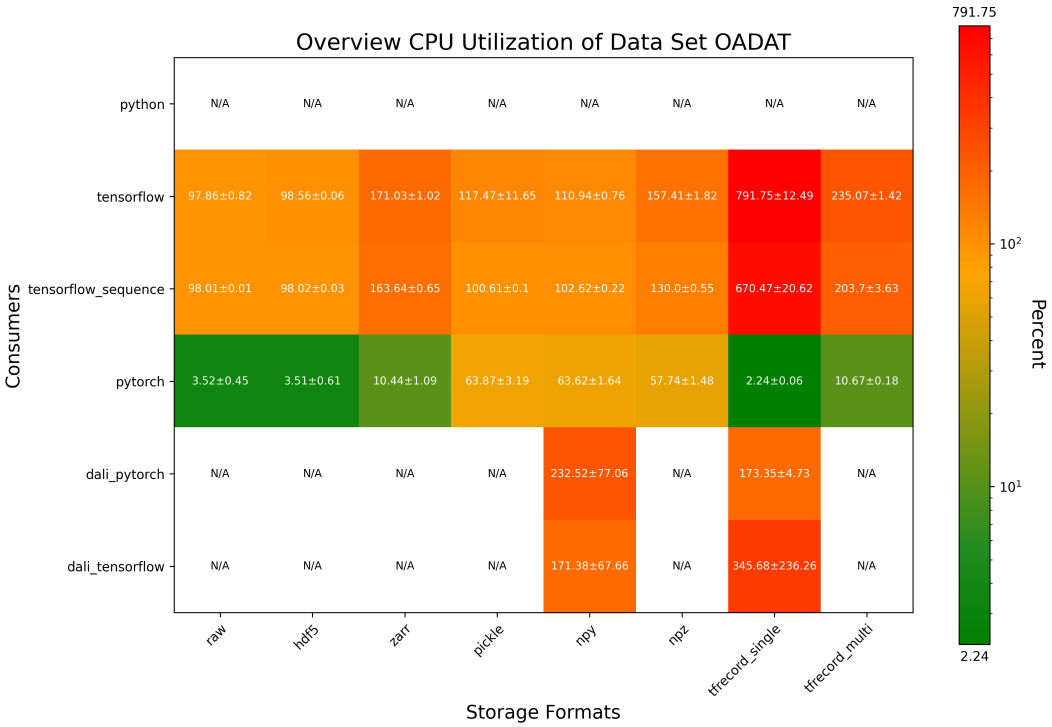


(b) `device=gpu, return_type=tensorflow`

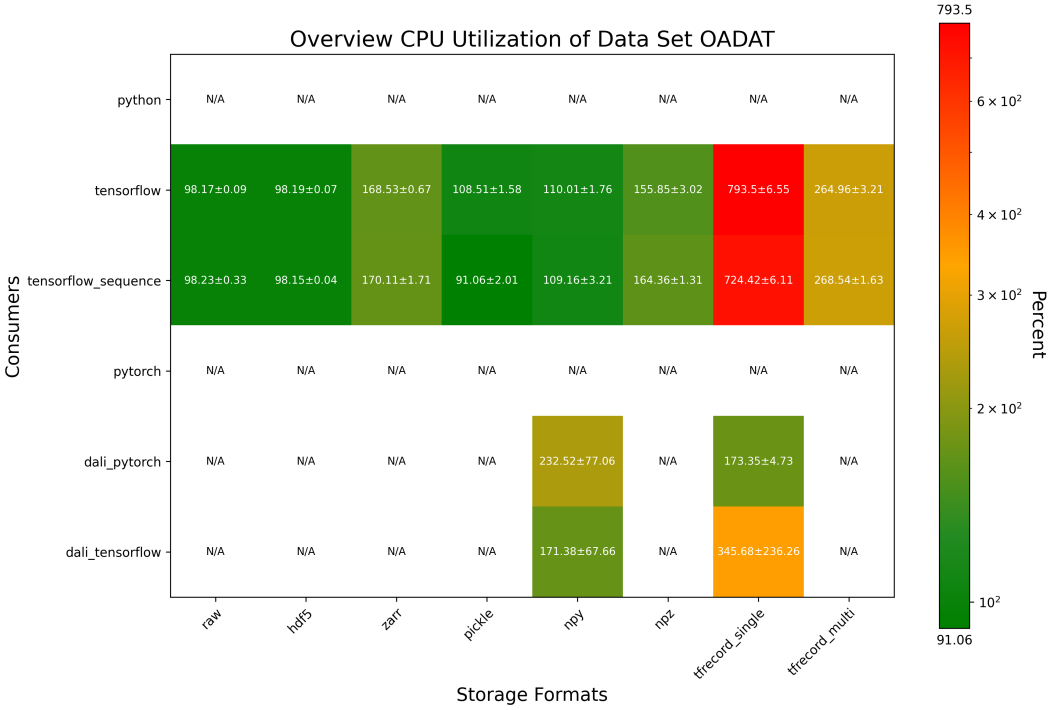


(c) `device=gpu, return_type=pytorch`

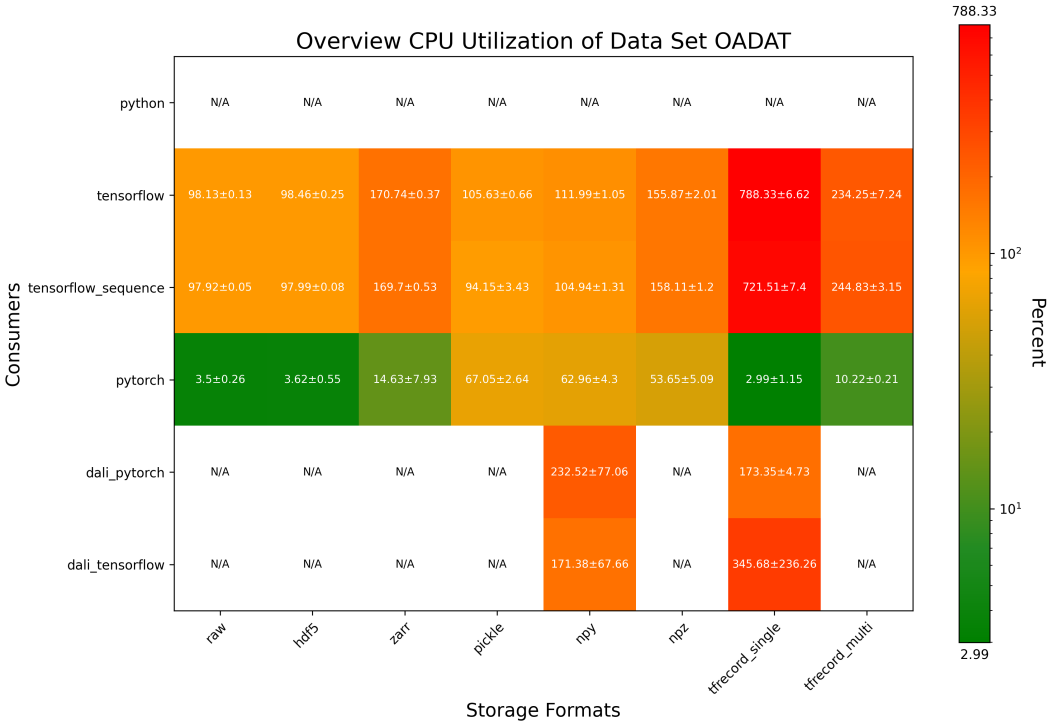
Figure A.2: Labeled runtime of OADAT on GPU-node `device=gpu`



(a) `device=gpu, return_type=python`

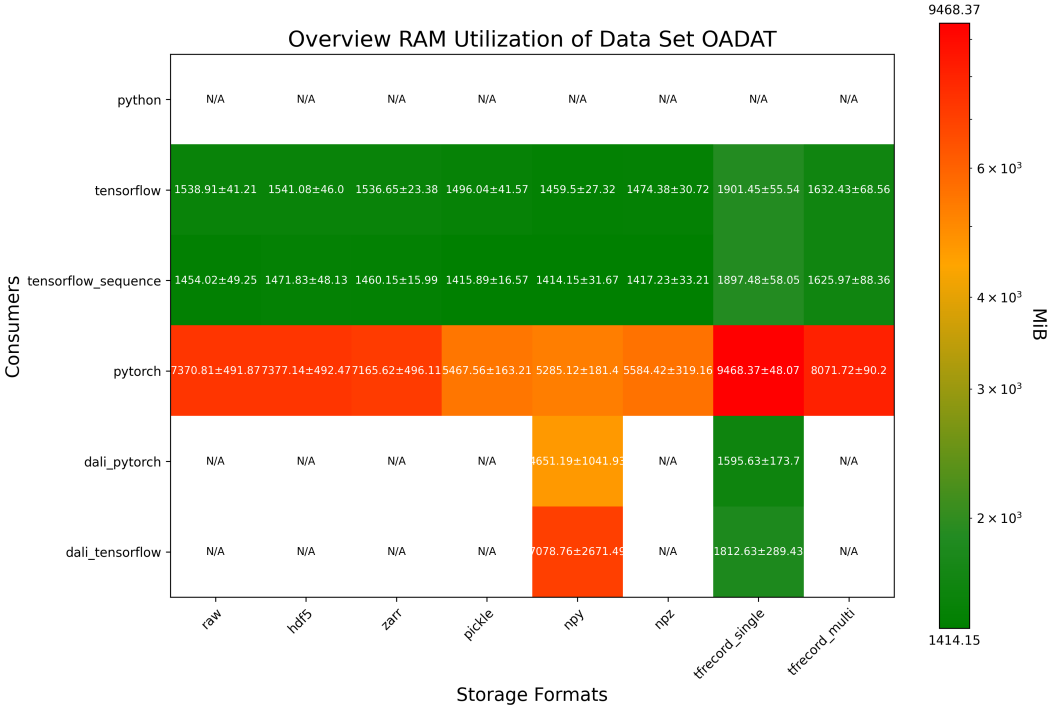


(b) `device=gpu`, `return_type=tensorflow`

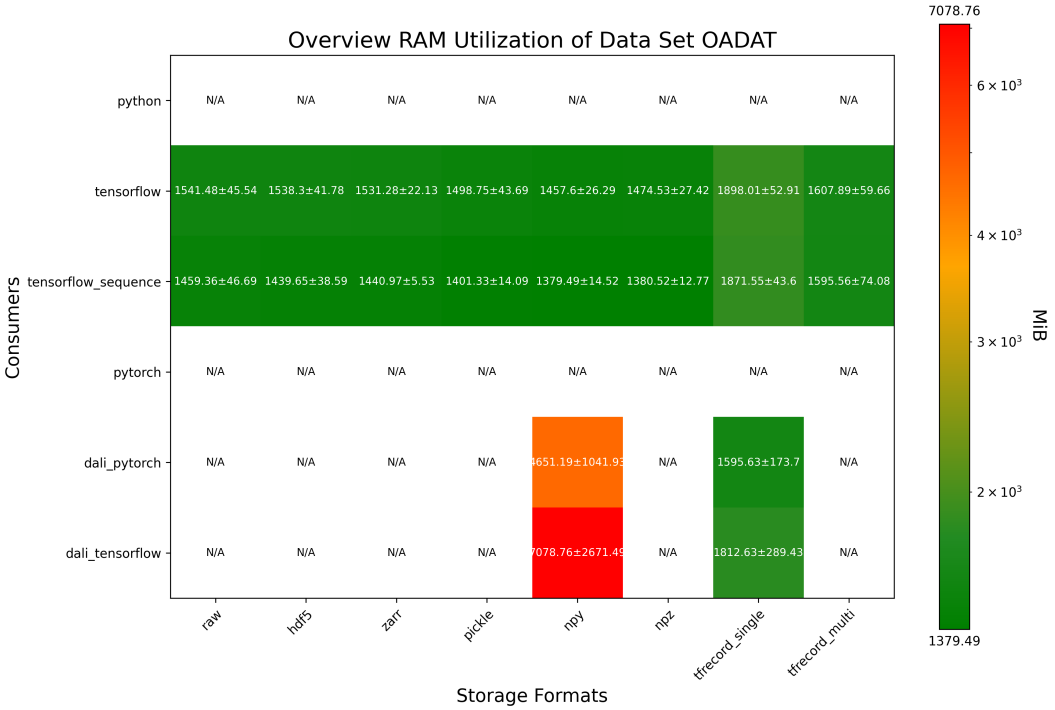


(c) `device=gpu`, `return_type=pytorch`

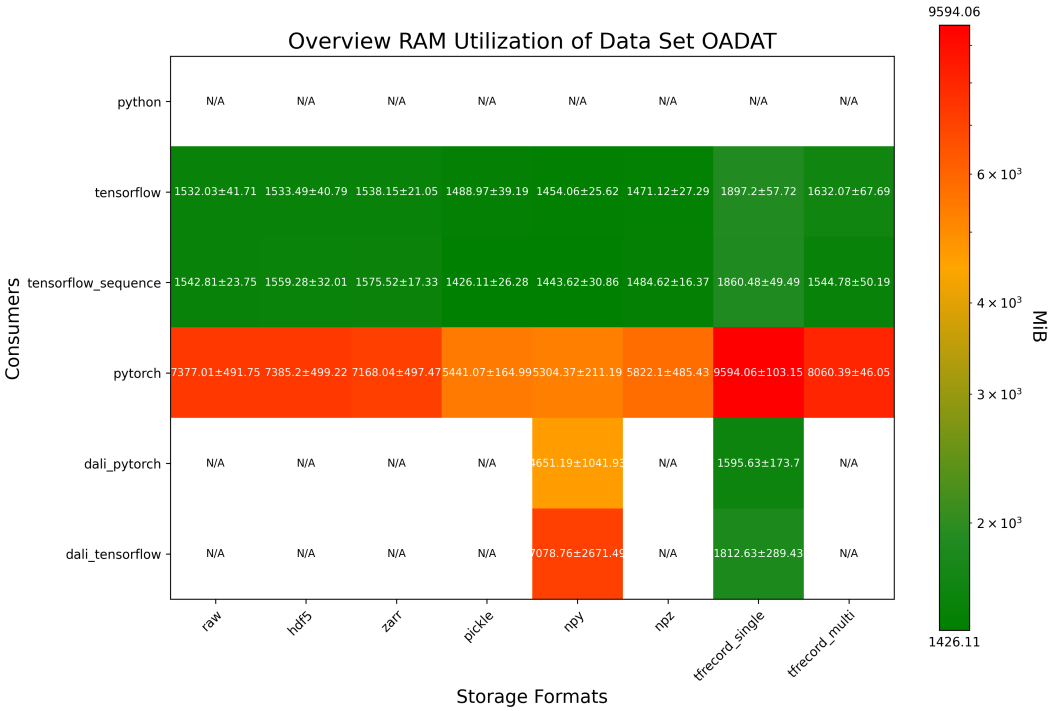
Figure A.3: Labeled CPU utilization of OADAT on GPU-node `device=gpu`



(a) `device=gpu, return_type=python`



(b) `device=gpu, return_type=tensorflow`



(c) `device=gpu`, `return_type=pytorch`

Figure A.4: Labeled RAM utilization of OADAT on GPU-node `device=gpu`

A.2 Results ICON Data Set

Configuration File A.2: ICON Data Set

```
dataset_name: 'ICON'
dataset_type: 'icon'
path: '<dataset-path>/
   ml_ecrad_ape_R2B05_myrunscript_1year_183min_ecrad_in'
filename: 'train_10p_10k_ss.h5'
key:
 - 'lwflx_up'
 - 'clc'
 - 'cosmu0'
batch_size: 32
num_threads: 12
```

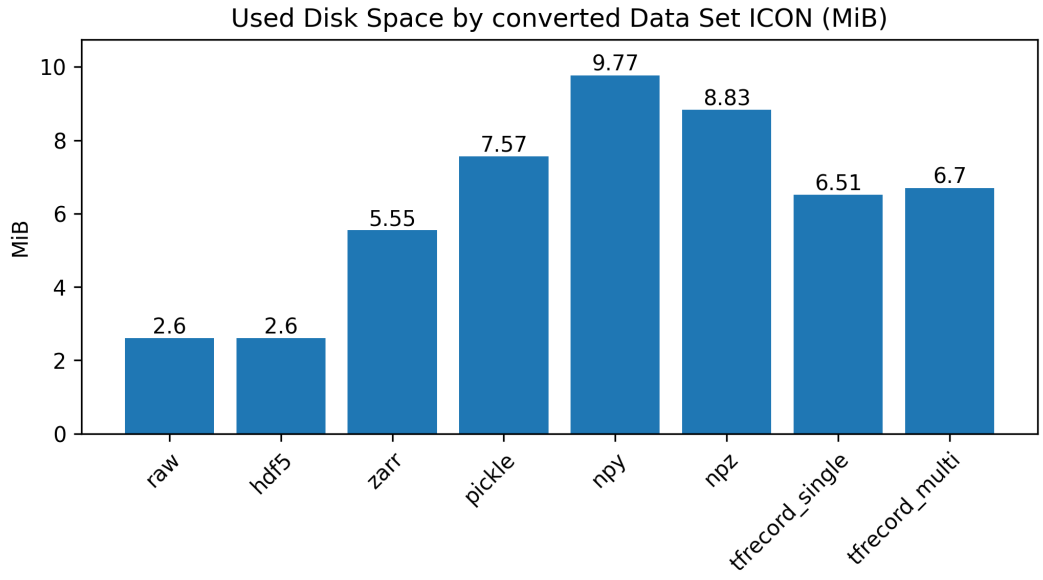
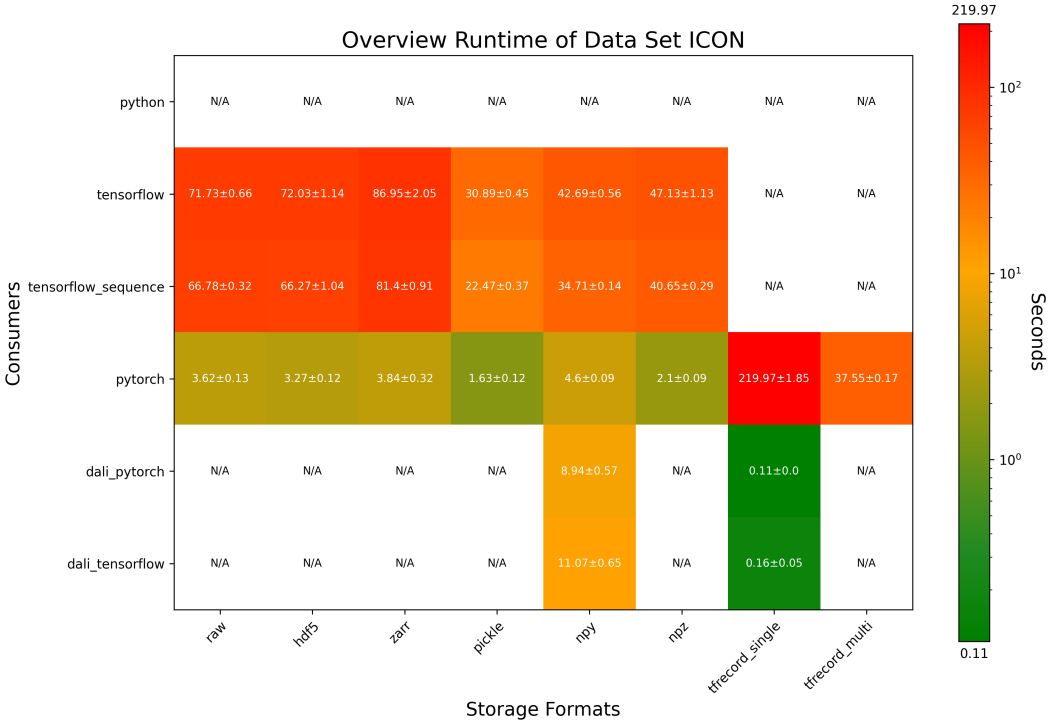
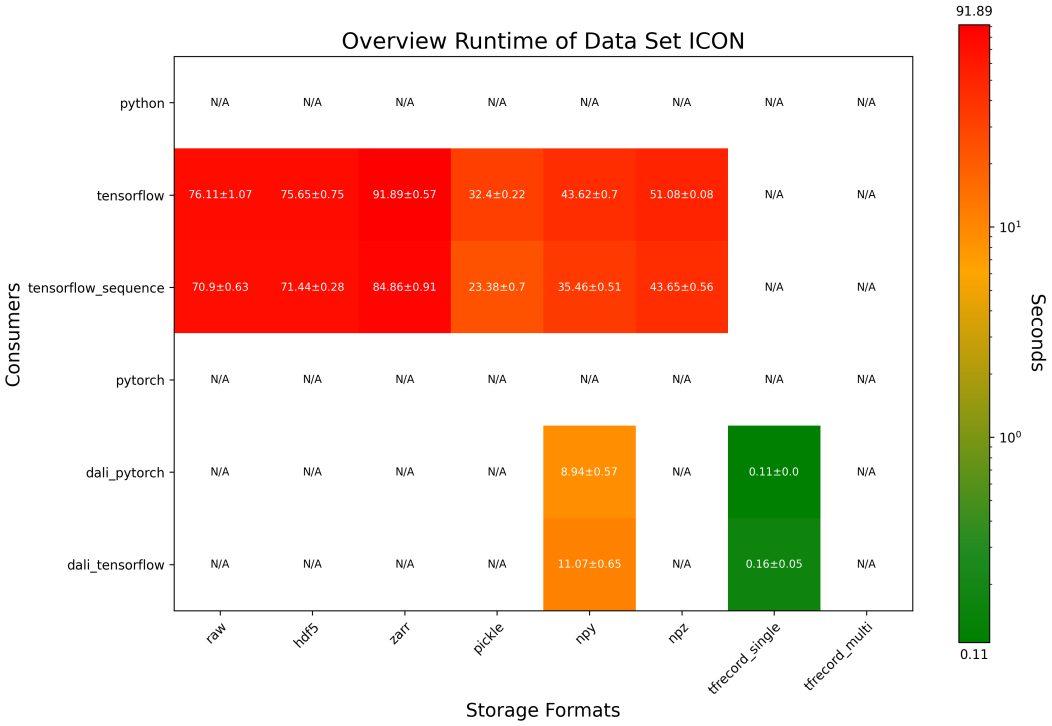


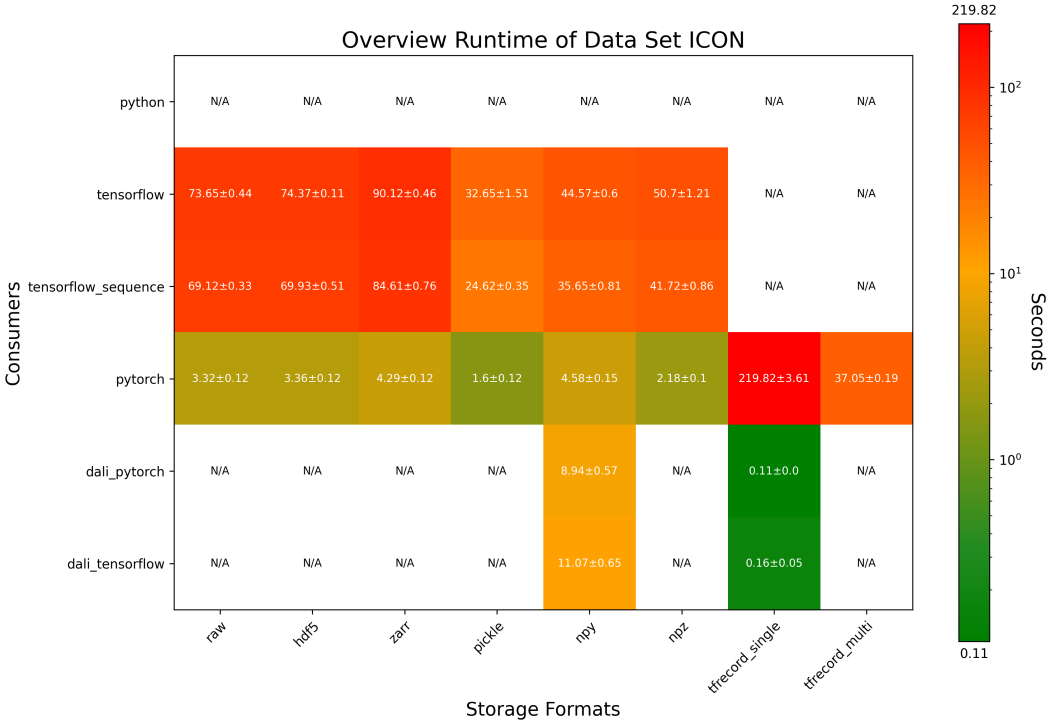
Figure A.5: Disk space used by conversions of ICON data set



(a) `device=gpu, return_type=python`

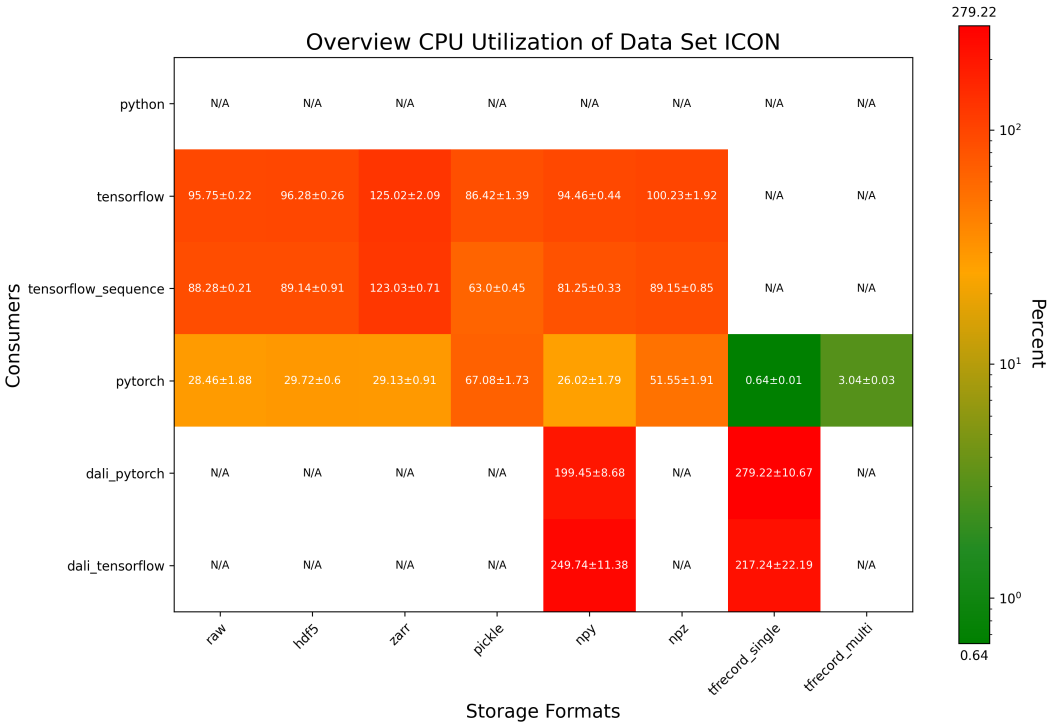


(b) `device=gpu, return_type=tensorflow`

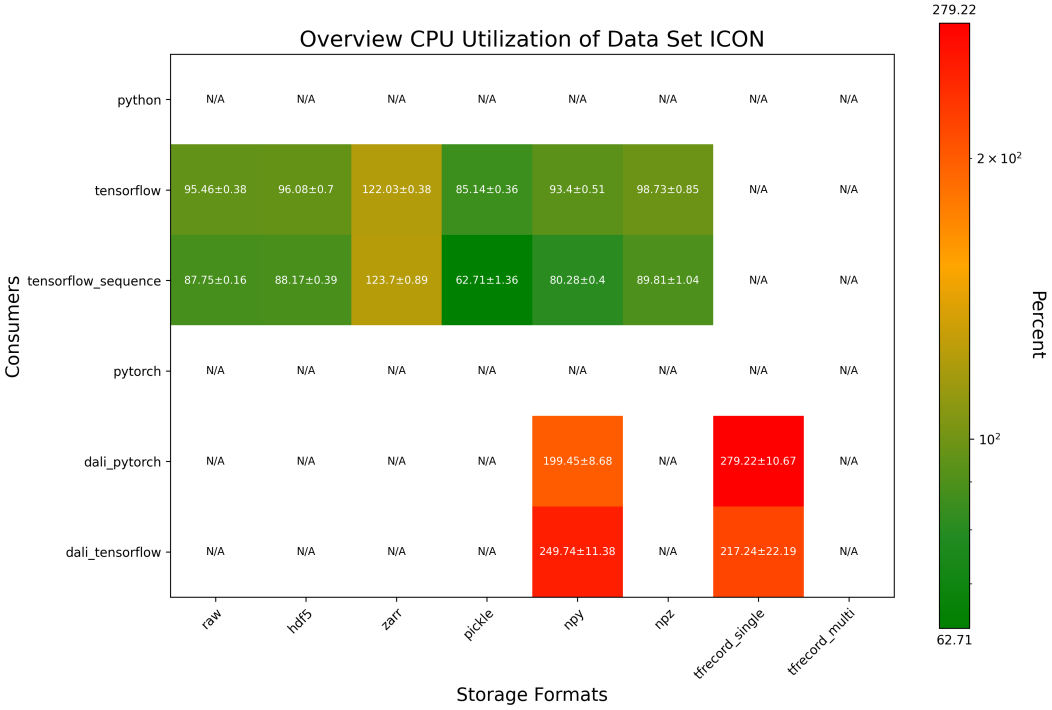


(c) `device=gpu, return_type=pytorch`

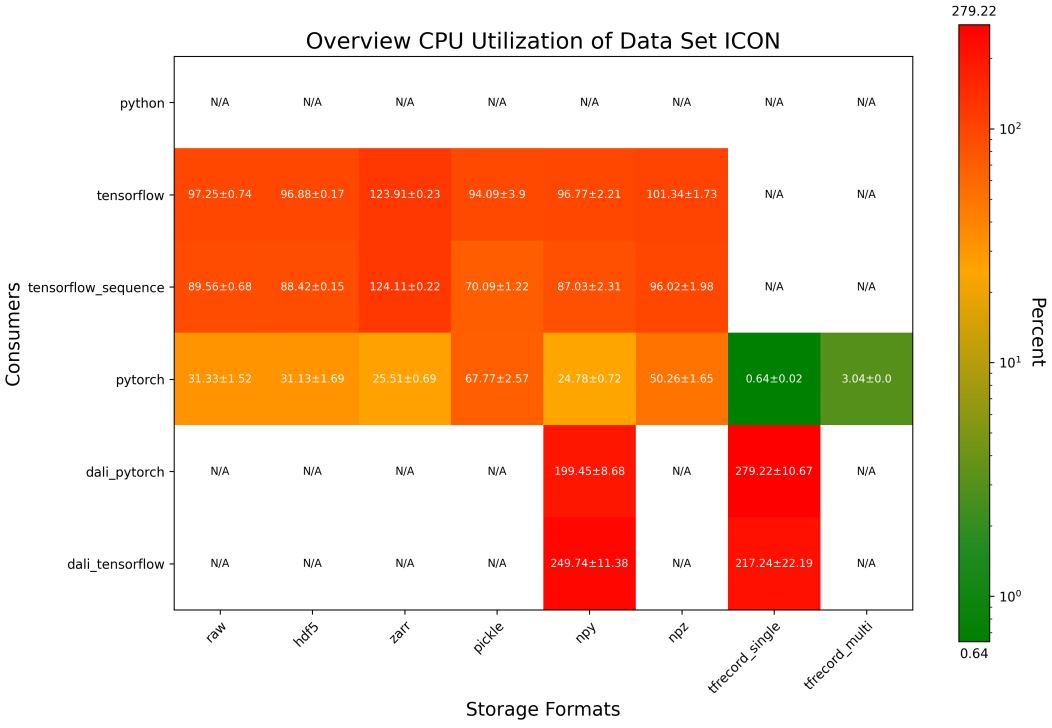
Figure A.6: Labeled runtime of ICON on GPU-node `device=gpu`



(a) `device=gpu, return_type=python`

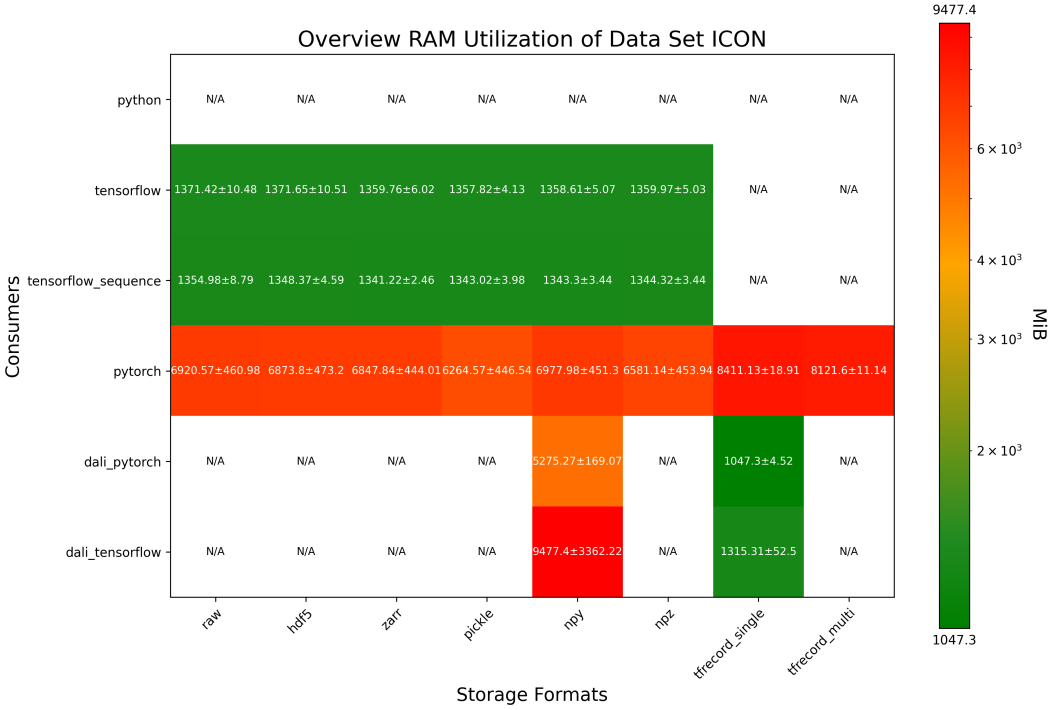


(b) `device=gpu, return_type=tensorflow`

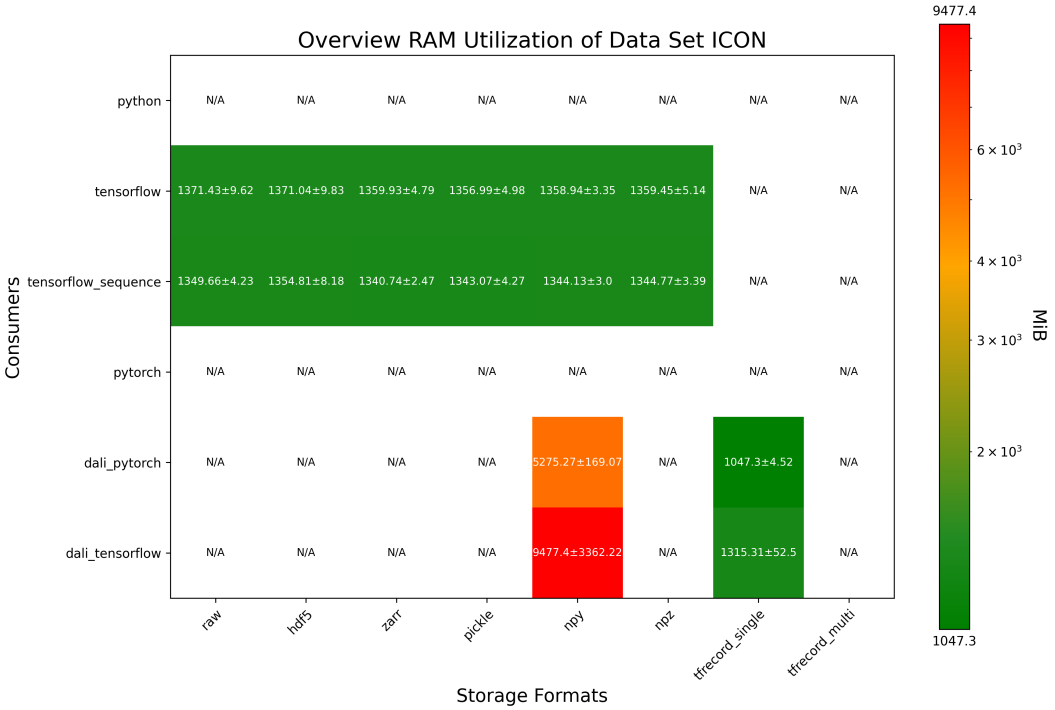


(c) `device=gpu, return_type=pytorch`

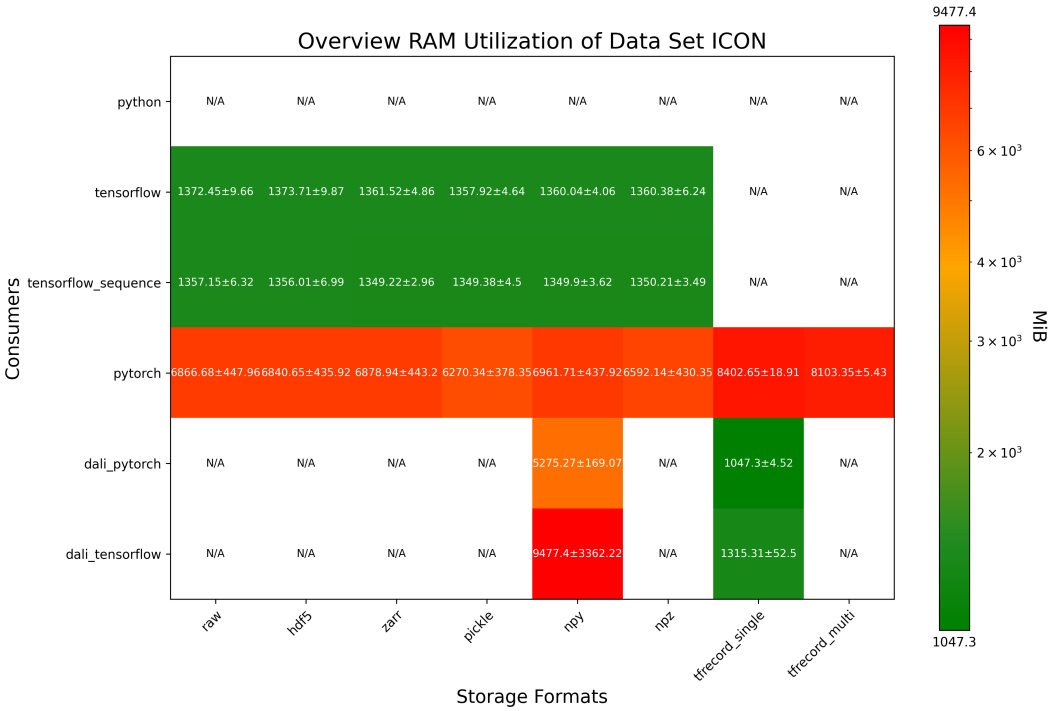
Figure A.7: Labeled CPU utilization of ICON on GPU-node `device=gpu`



(a) `device=gpu, return_type=python`



(b) `device=gpu, return_type=tensorflow`



(c) `device=gpu`, `return_type=pytorch`

Figure A.8: Labeled RAM utilization of ICON on GPU-node `device=gpu`

A.3 Results ILSVRC 2012 Data Set

Configuration File A.3: ILSVRC 2012 Data Set

```
dataset_name: 'ImageNet'
dataset_type: 'imagenet'
path: '<dataset-path>/ILSVRC2012_img_val'
amount: 2048
batch_size: 32
num_threads: 12
```

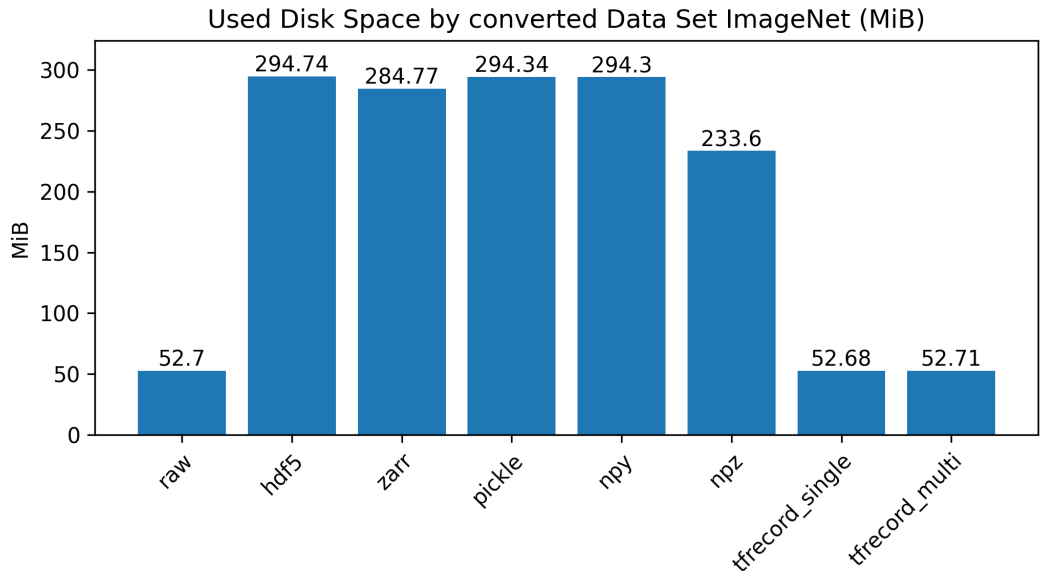
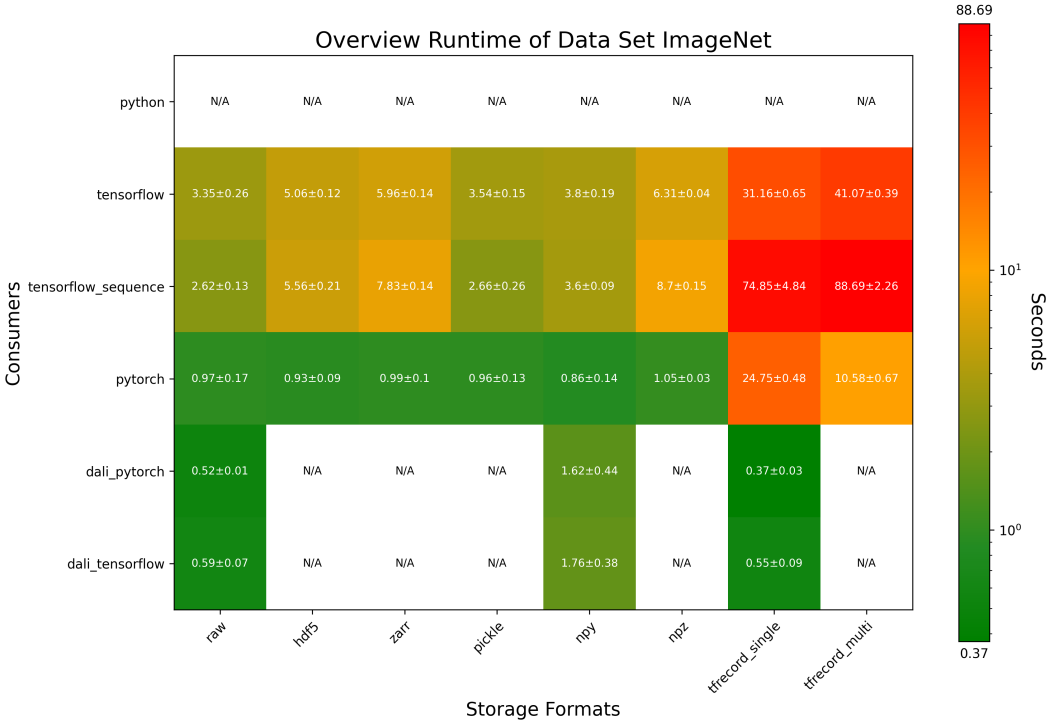
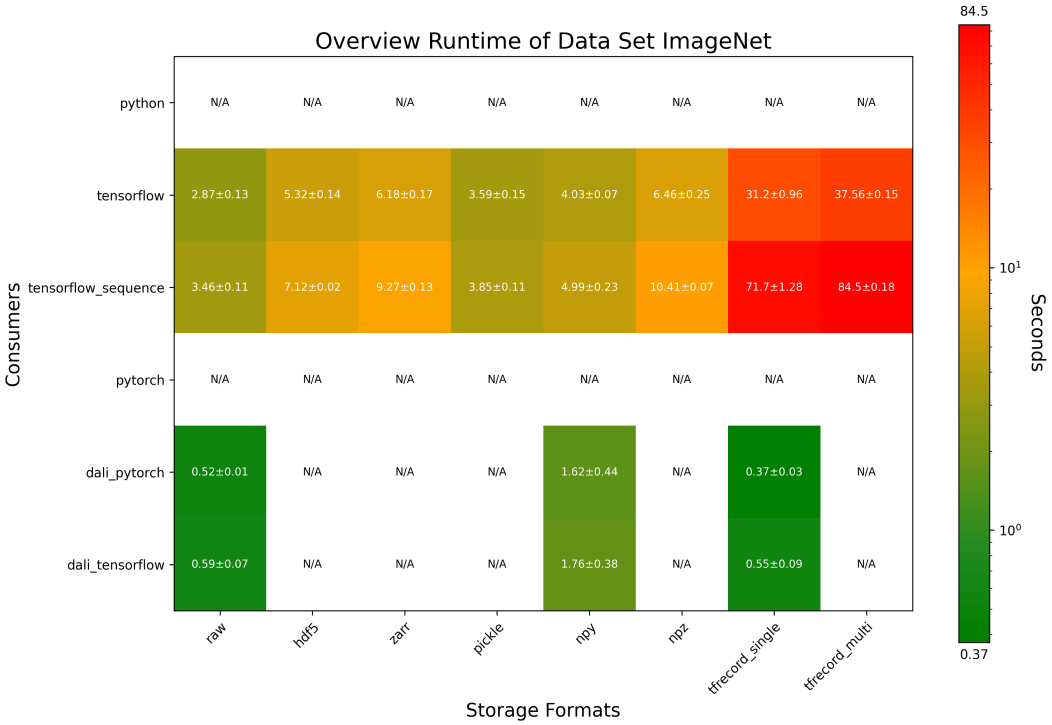


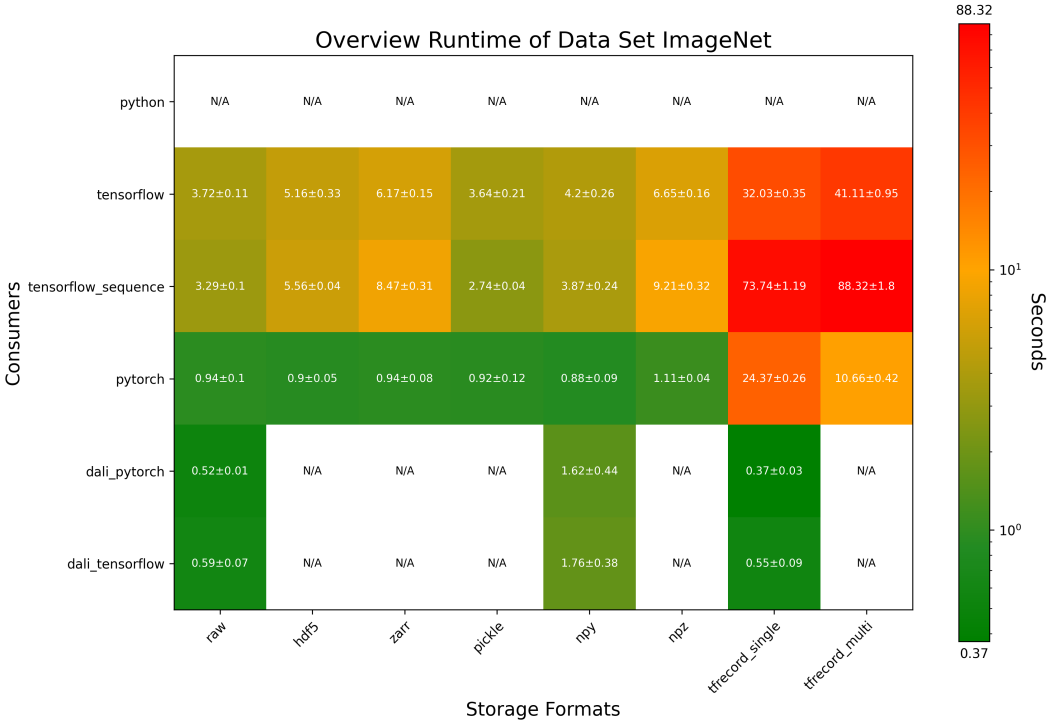
Figure A.9: Disk space used by conversions of ILSVRC 2012 data set



(a) `device=gpu, return_type=python`

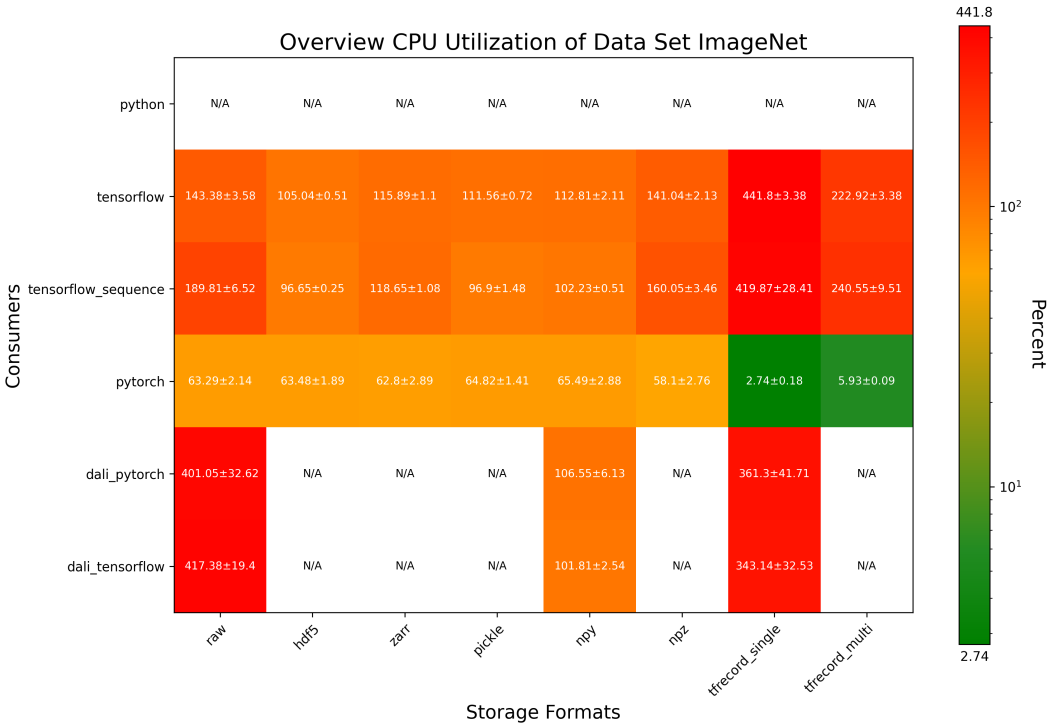


(b) `device=gpu, return_type=tensorflow`

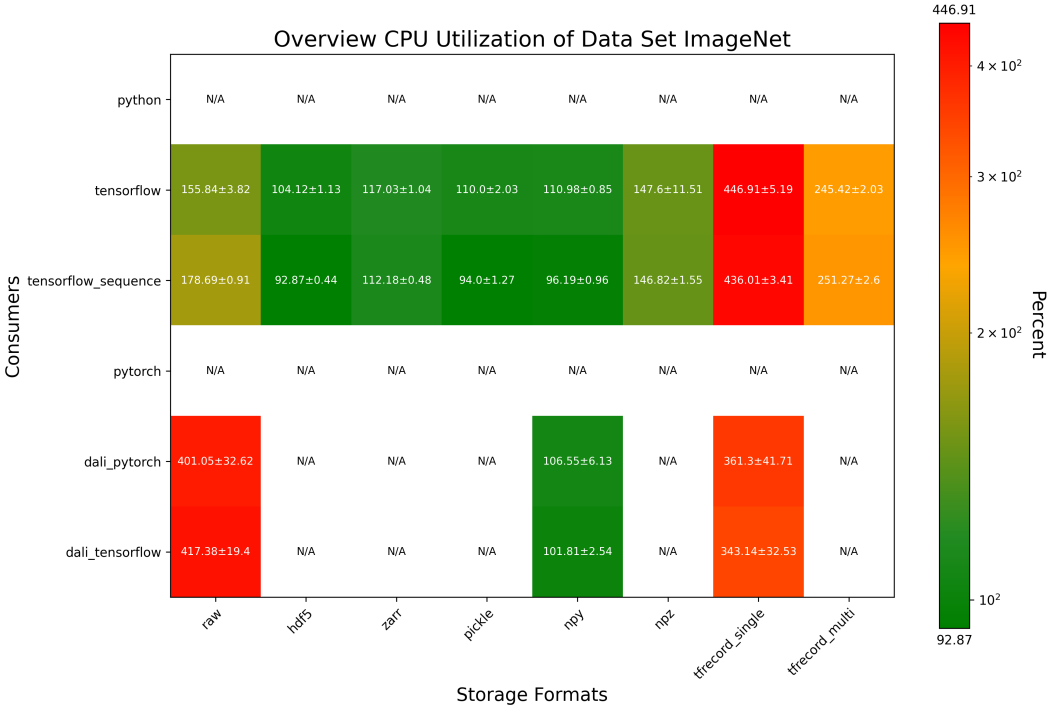


(c) `device=gpu, return_type=pytorch`

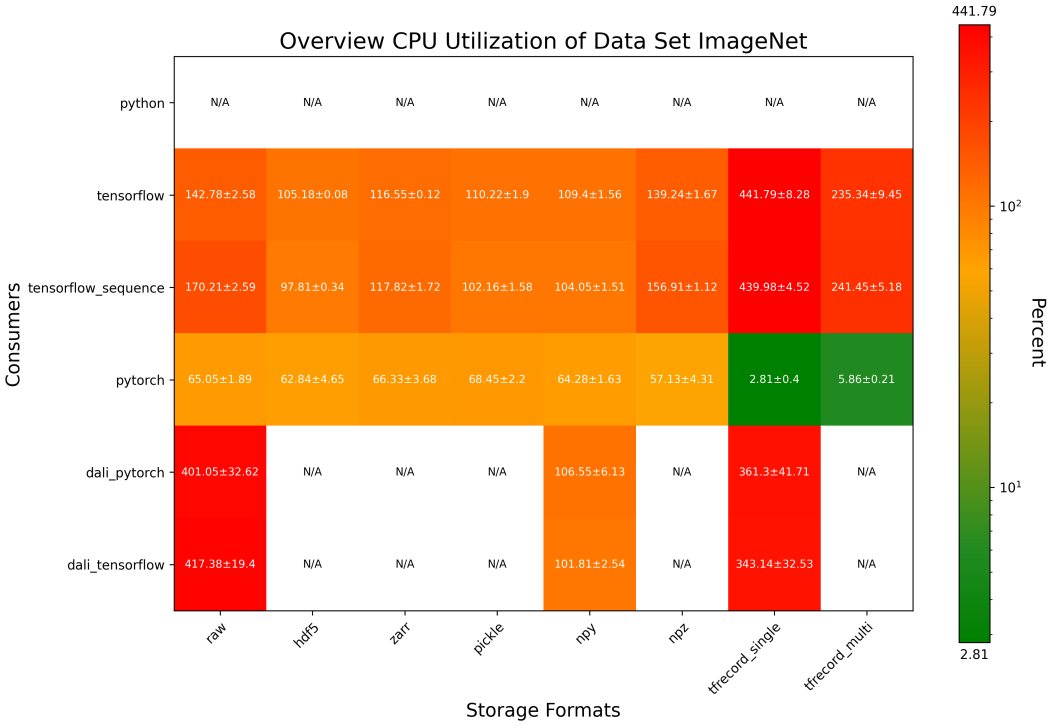
Figure A.10: Labeled runtime of ILSVRC 2012 on GPU-node `device=gpu`



(a) `device=gpu, return_type=python`

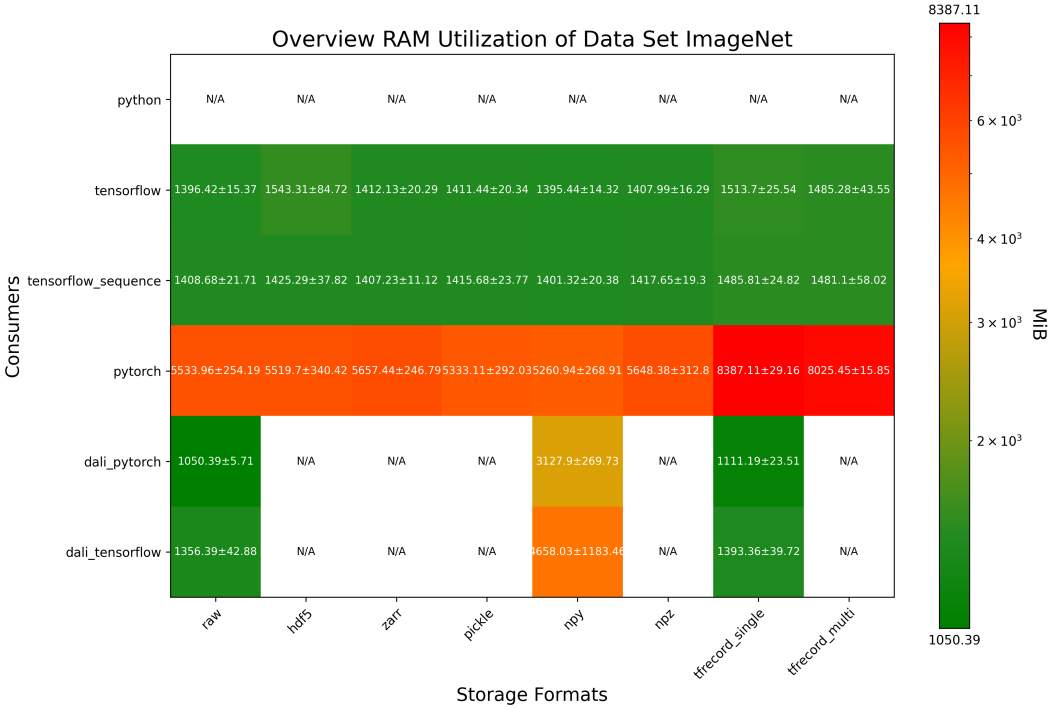


(b) `device=gpu, return_type=tensorflow`

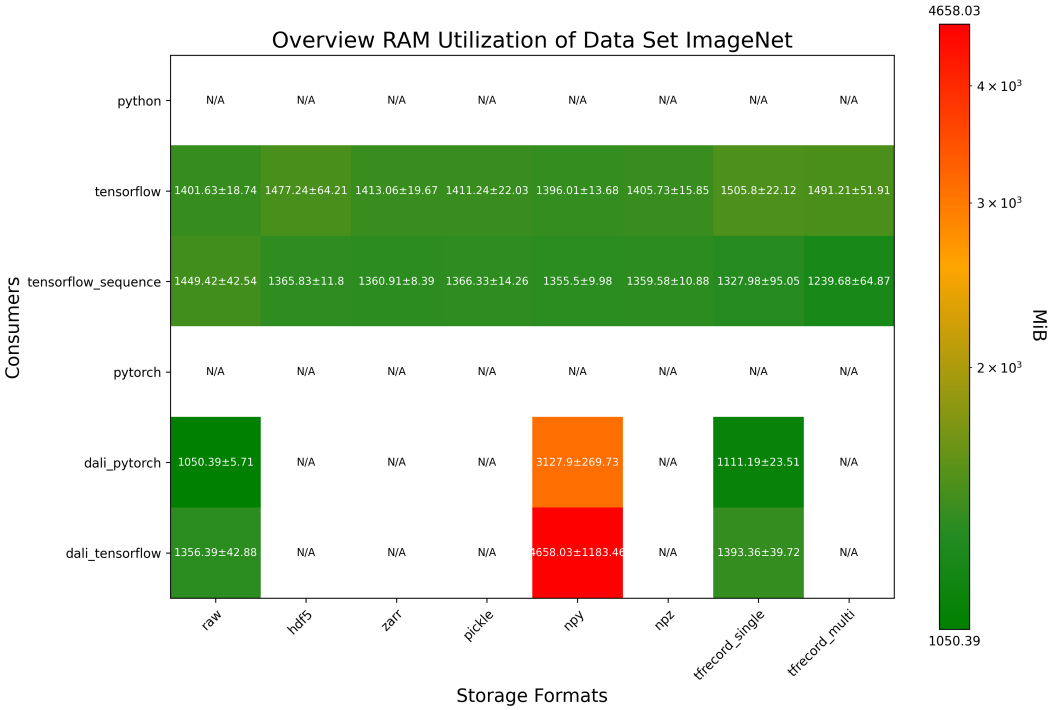


(c) `device=gpu, return_type=pytorch`

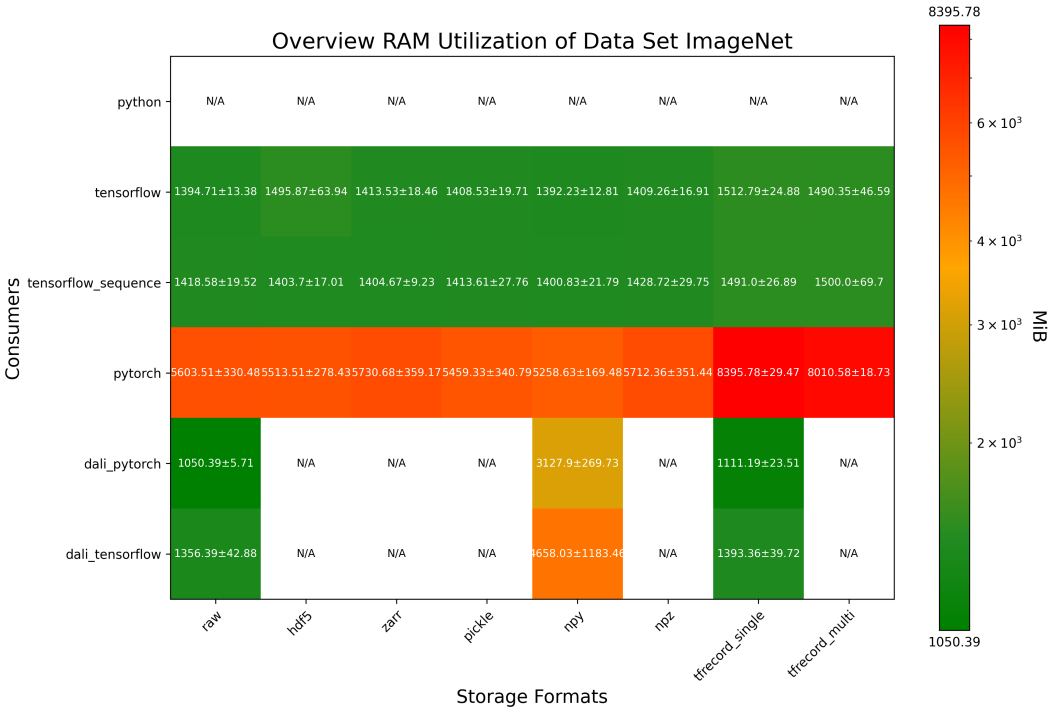
Figure A.11: Labeled CPU utilization of ILSVRC 2012 on GPU-node `device=gpu`



(a) `device=gpu, return_type=python`



(b) `device=gpu, return_type=tensorflow`



(c) `device=gpu`, `return_type=pytorch`

Figure A.12: Labeled RAM utilization of ILSVRC 2012 on GPU-node `device=gpu`

A.4 Results ShapeNet Data Set

Configuration File A.4: ShapeNet Data Set

```yaml
dataset_name: 'ShapeNet'
dataset_type: 'shapenet'
path: '<dataset-path>'
key:
 - 'tower'
 - 'knife'
batch_size: 16
num_threads: 12
```

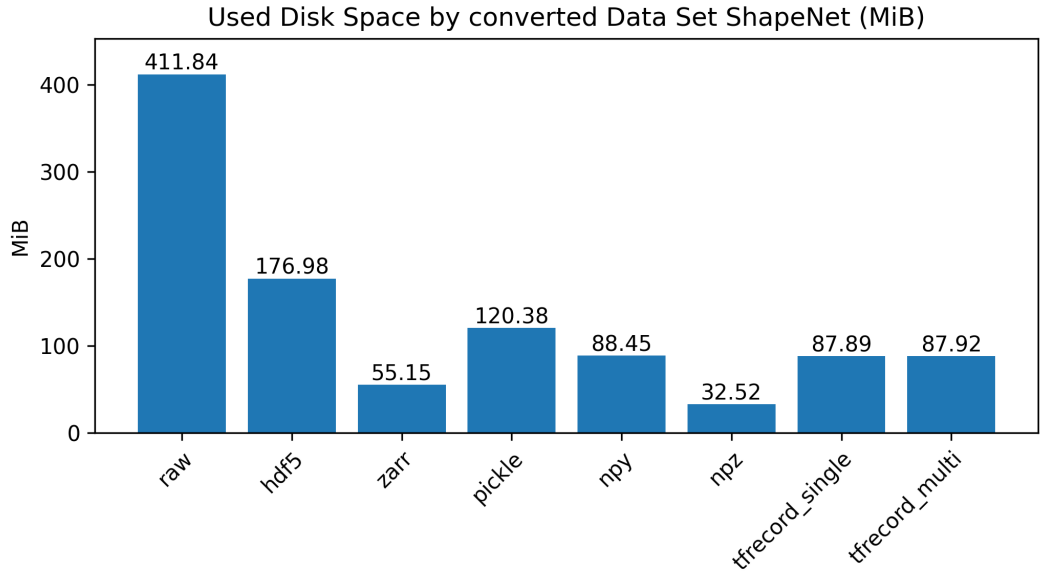
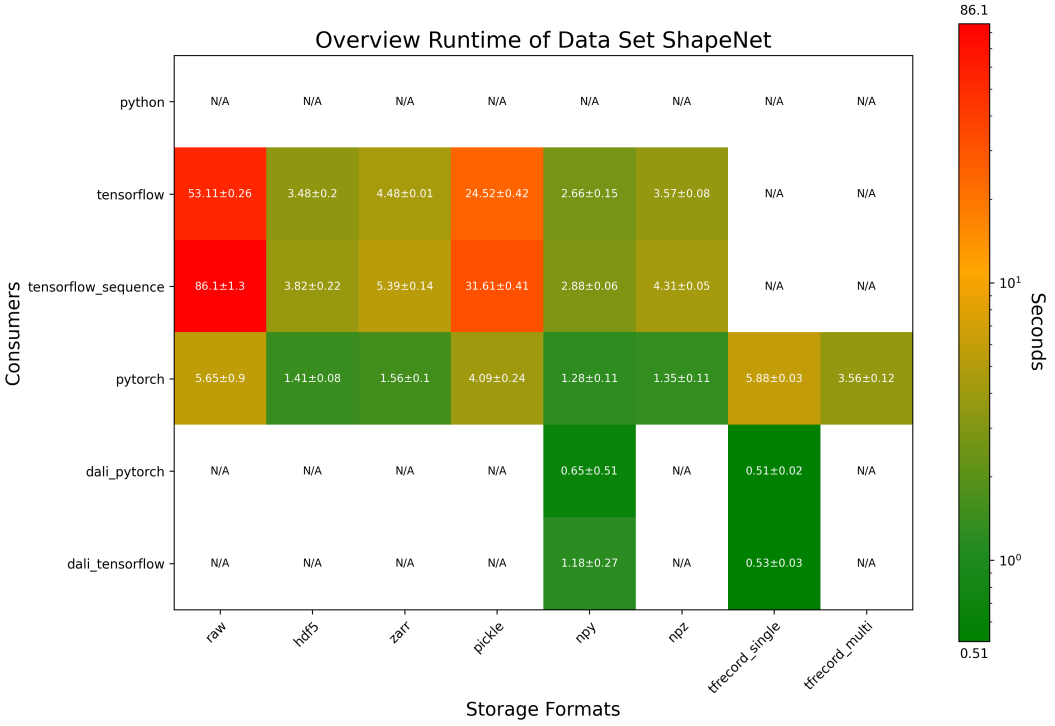
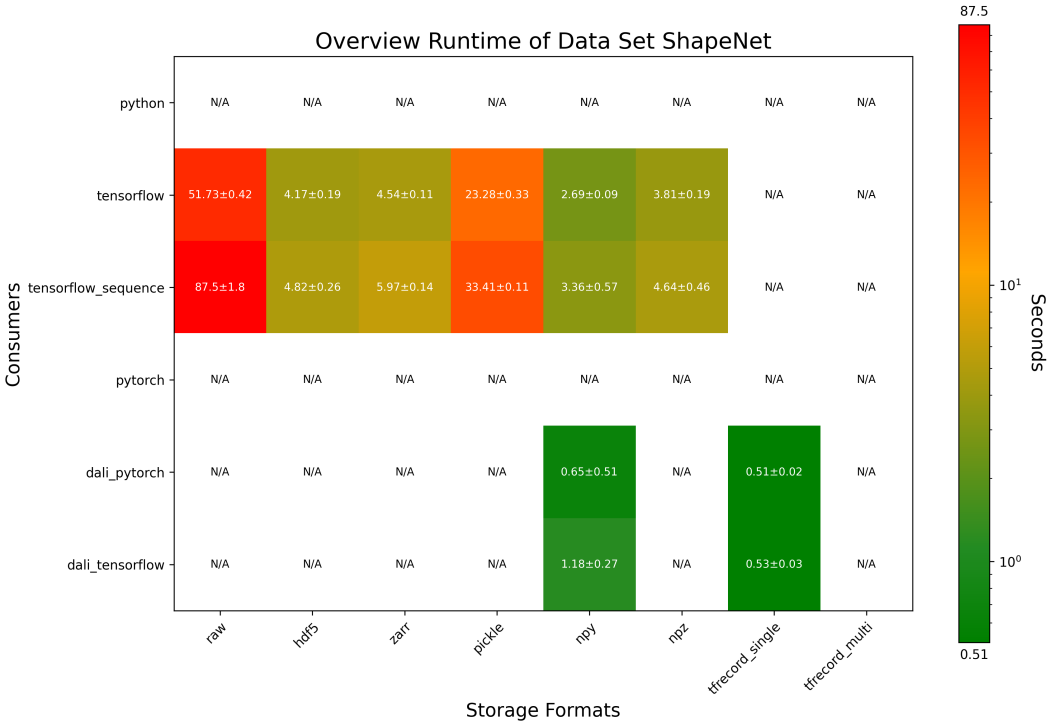


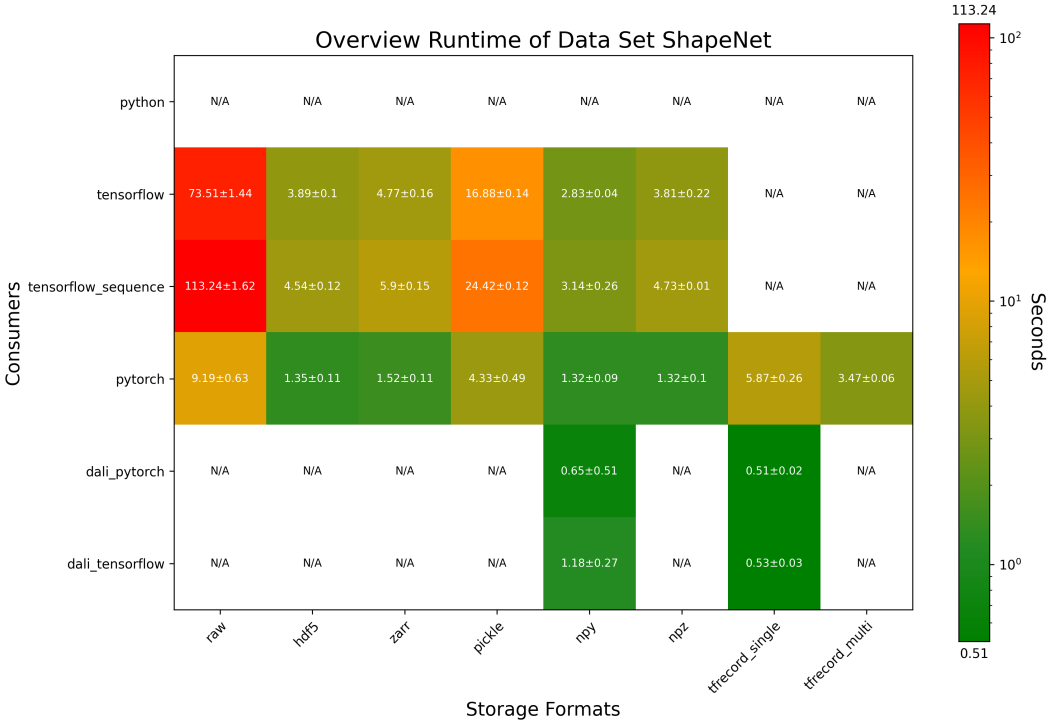
Figure A.13: Disk space used by conversions of ShapeNet data set



(a) `device=gpu, return_type=python`

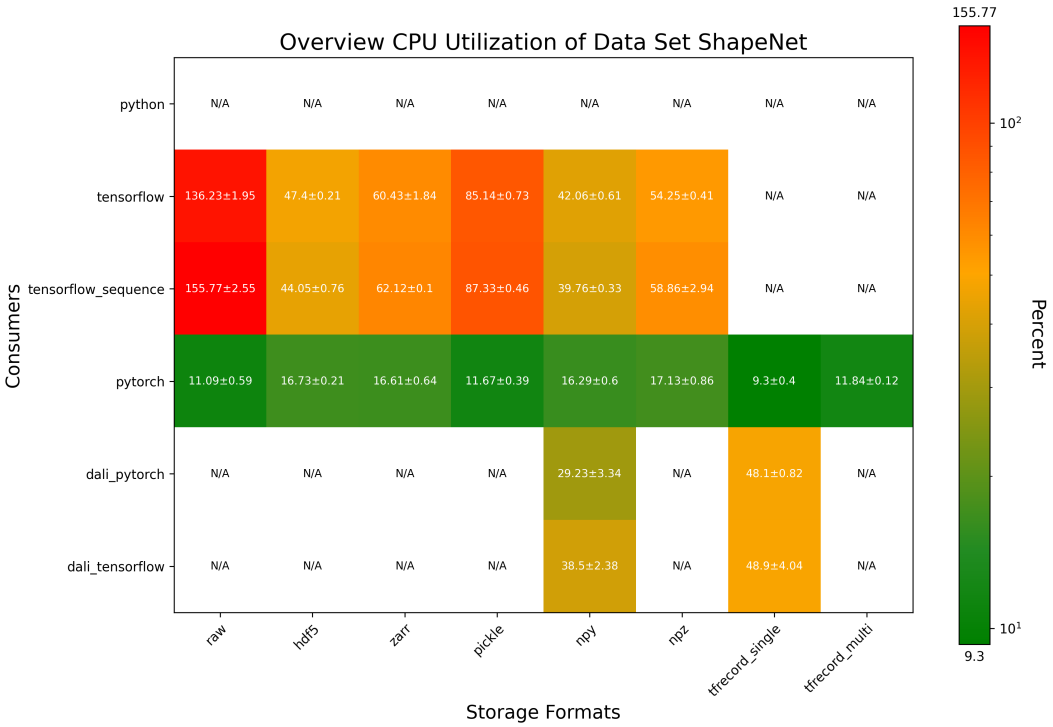


(b) `device=gpu, return_type=tensorflow`

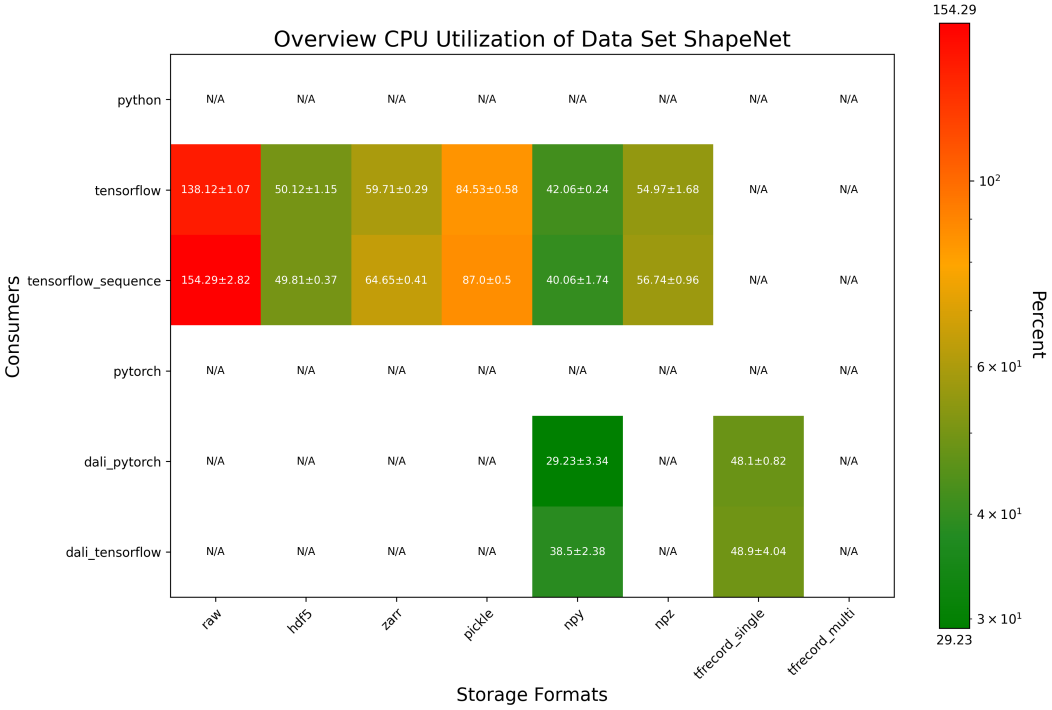


(c) `device=gpu, return_type=pytorch`

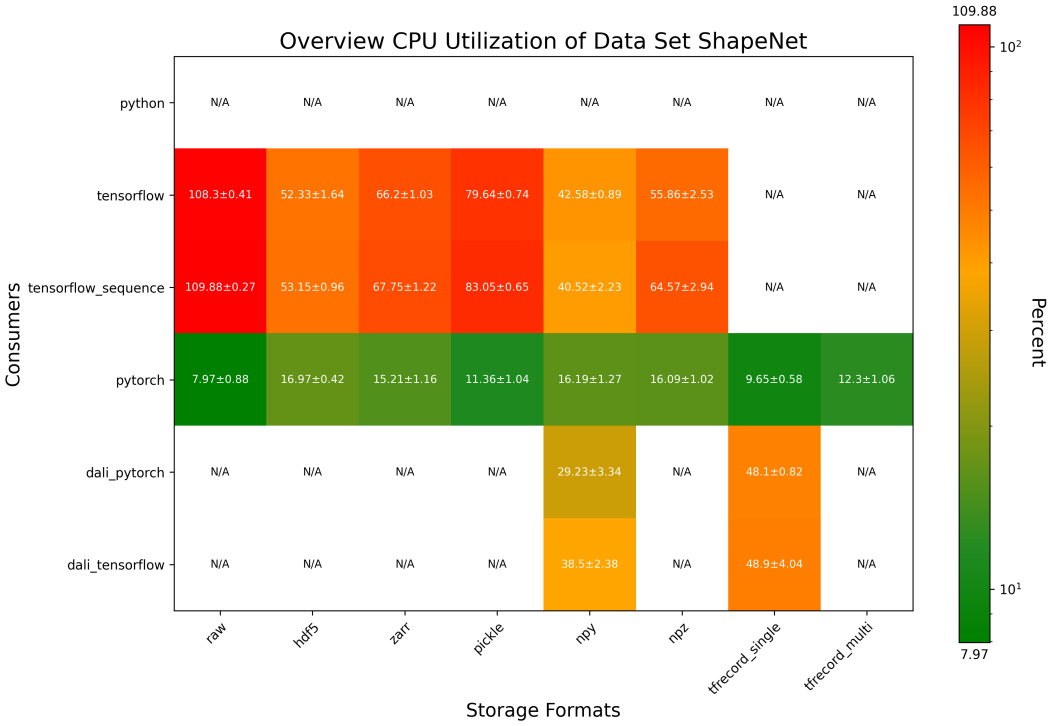
Figure A.14: Labeled runtime of ShapeNet on GPU-node `device=gpu`



(a) `device=gpu, return_type=python`

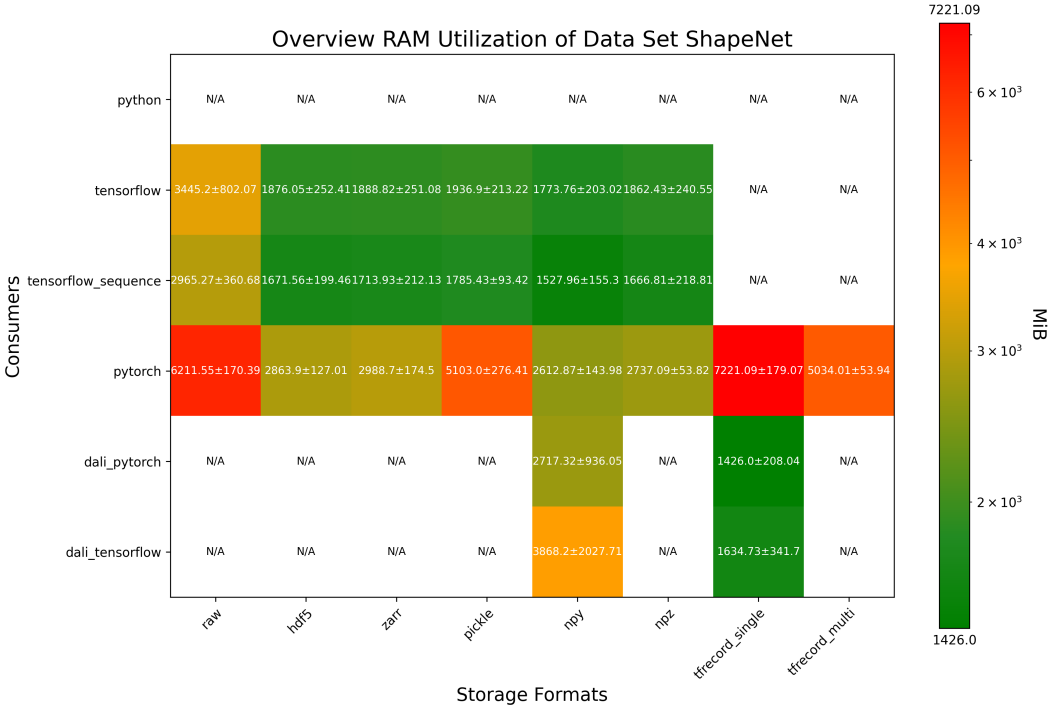


(b) `device=gpu, return_type=tensorflow`

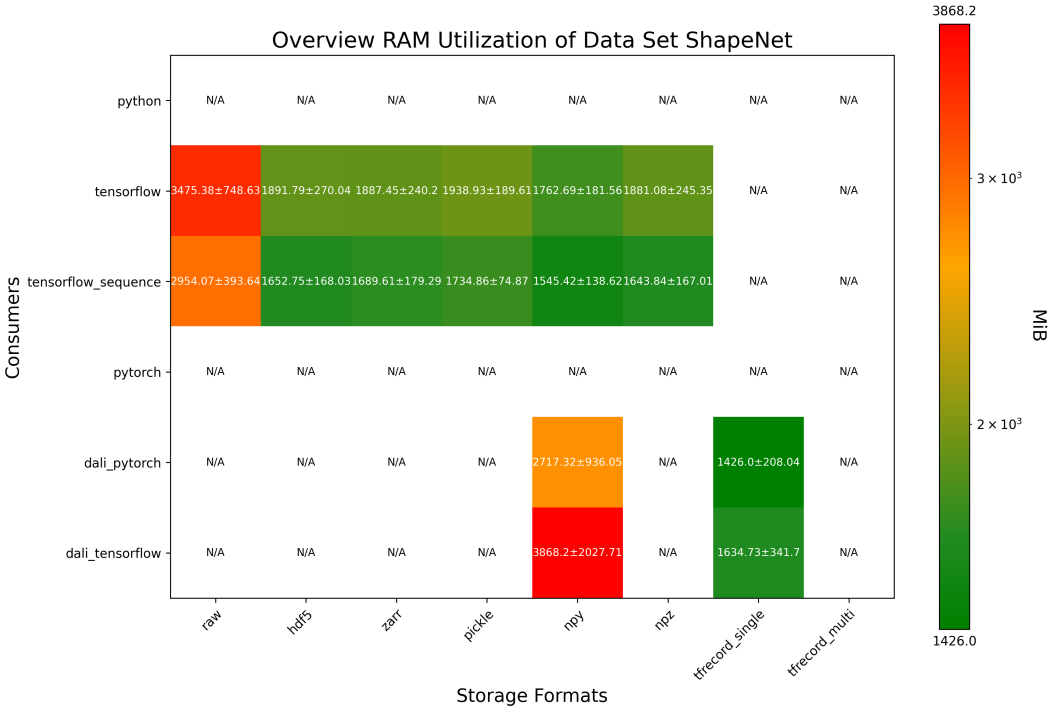


(c) `device=gpu, return_type=pytorch`

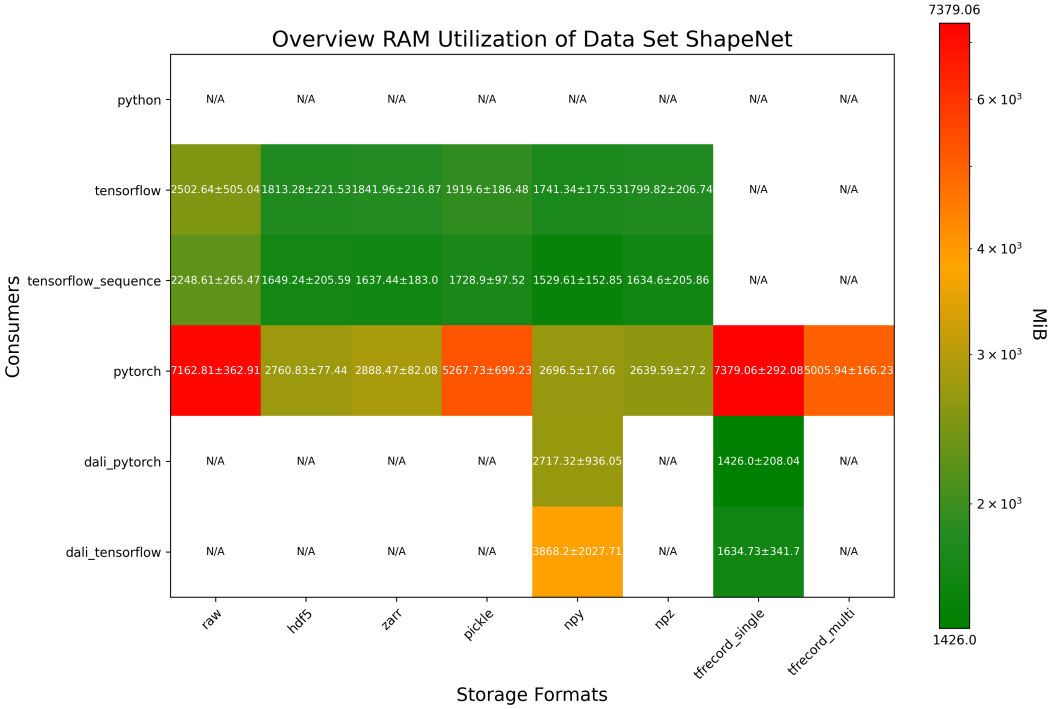
Figure A.15: Labeled CPU utilization of ShapeNet on GPU-node `device=gpu`



(a) `device=gpu, return_type=python`



(b) `device=gpu, return_type=tensorflow`



(c) `device=gpu`, `return_type=pytorch`

Figure A.16: Labeled RAM utilization of ShapeNet on GPU-node `device=gpu`