



# Input Data Pipeline Analysis of TensorFlow Models

FNU SACHIN



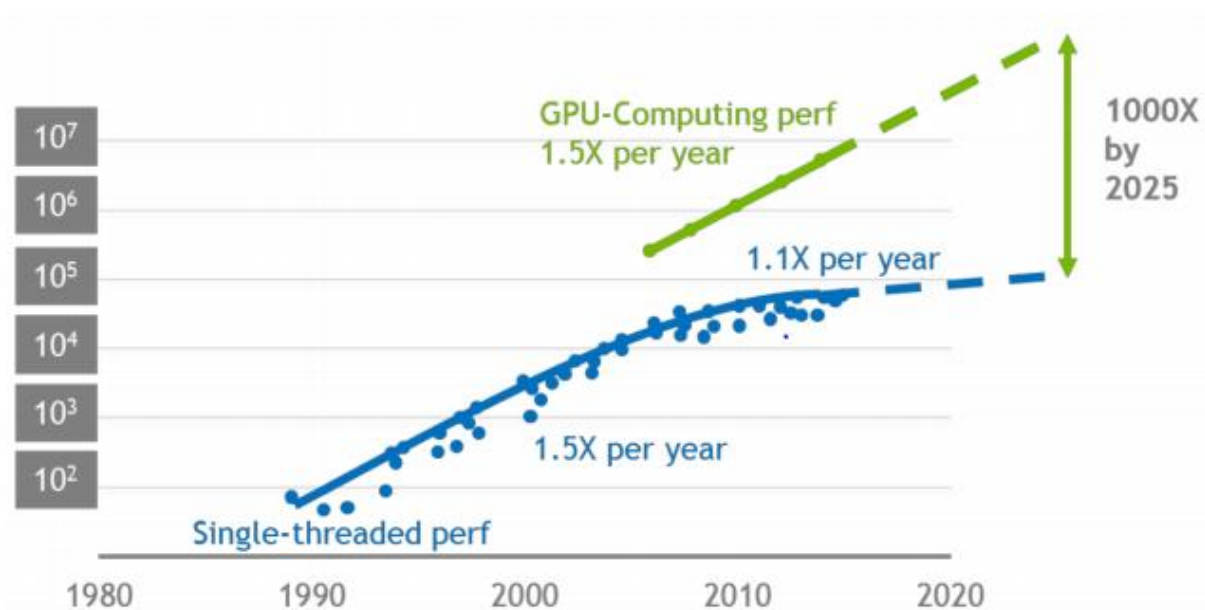


# Overview

- To understand how deep learning models interact with underlying hardware such as storage systems, CPUs, GPUs.
- To identify bottlenecks in the data loading path for deep learning models.
- To identify the potential solution/optimizations to resolve the bottlenecks and to enhance model performance.
- To understand the emerging storage solutions such as SmartSSD that comes with integrated Xilinx FPGA to accelerate data intensive operations.
- To identify ways to integrate SmartSSD solution with existing deep learning models.

# Background

Increasing difference in computing performance of GPU and CPU and infeasibility of CPU scaling result in increasing preprocessing stalls. This trend motivates us to consider offloading preprocessing onto an accelerator.



Source: Nvidia

- If CPU pre-processing speed is less than GPU ingestion speed, GPU starves for new pre-processed data.
- GPU performance has been improving 1.5x every year. However, CPU performance improvement has been saturated for a decade.
- Larger performance gap means that GPU will be idle for longer amount of time and the entire DL training will be stalled.

## Basic Definitions

- *Batch Size*: It represents the number of samples of input dataset that are processed before the model is updated.
- *Epoch*: It represents the number of times the learning algorithm will be executed to work through entire dataset.
- *ETL*: It stands for Extract, Transform and Load. It defines the steps of data loading process in deep learning framework.
- *RGB Image*: An RGB image, contains three individual planes of colors red, blue and green. The color of each pixel is determined by the combination of the red, green, and blue intensities stored in each color plane at the pixel's location.



# Presentation Outline

- *Data Pipeline*
- *TensorFlow Input Pipeline*
- *Preprocessing Steps*
- *Deep Residual Models*
- *Experimental Setup*
- *Experimental Analysis*
- *Optimizations*
- *Conclusion*

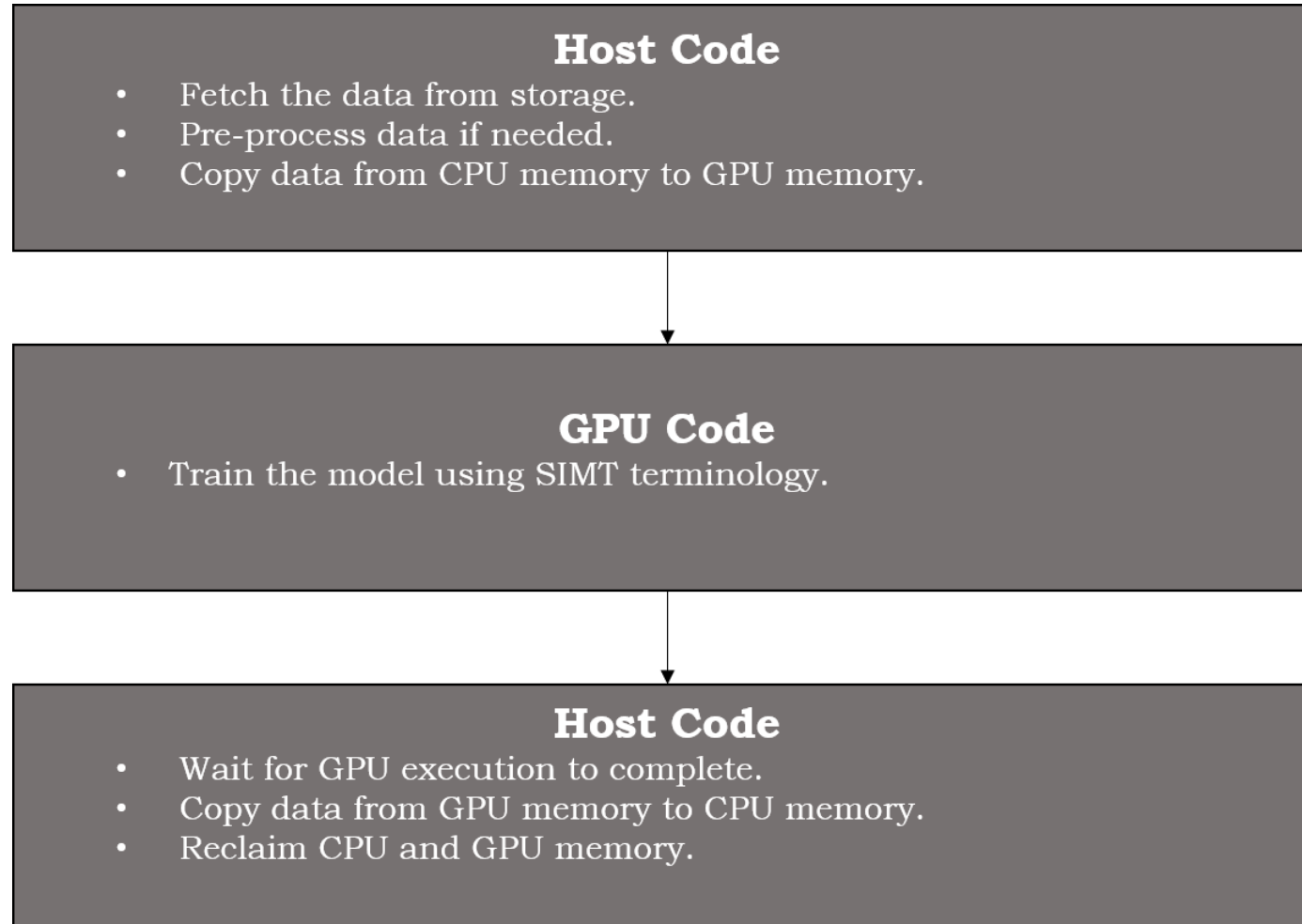
01



## Data Pipeline

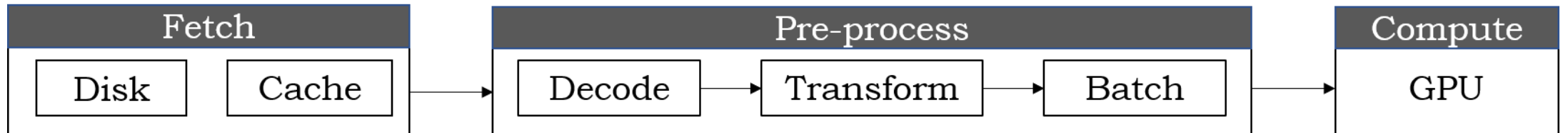
- Introduction to the concept of data pipeline.
- Various pipeline stalls.

# Data Movement between Host and Device



# Data Pipeline Stages

- Fetching data from local/remote storage. Depends on storage media. (I/O bound)
- Pre-processing the fetched data. Depends upon available CPU cores (CPU bound).
- Processing final data using compute cluster. (GPU bound)





# Sequential vs Pipeline Data Model

- Sequential Model waits till one execution unit completes its execution.
- Pipeline Model achieves efficiency by overlapping various execution unit's operations.

**Sequential Model**

Host to Device Engine	0		
Kernel Engine		0	
Device to Host Engine			0

**Pipeline Model**

Host to Device Engine	1	2	3	4							
Kernel Engine		1	2	3	4						
Device to Host Engine			1	2	3	4					

time →

# Data Pipeline Stalls

Due to several stages in pipeline, delay at any stage will cause stall in pipeline.

- *Fetch stall* occurs due to delay in fetching the data from storage media. These types of stalls make data intensive process I/O bound.
- *Prep stall* occurs due to delay in pre-processing of the fetched data. These types of stalls make data intensive process CPU bound.

Fetch and Prep stall are collectively called *Data stalls*.



# Presentation Outline

- ✓ *Data Pipeline*
  - *TensorFlow Input Pipeline*
  - *Preprocessing Steps*
  - *Deep Residual Models*
  - *Experimental Setup*
  - *Experimental Analysis*
  - *Optimizations*
  - *Conclusion*

02

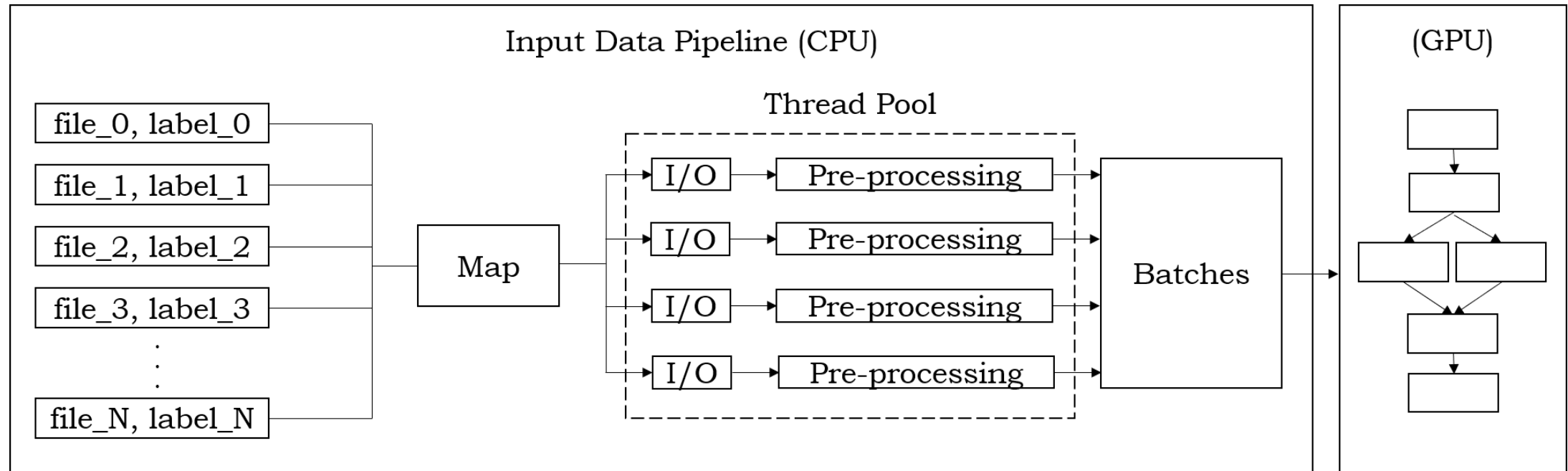


## TensorFlow Data Pipeline

- How TensorFlow implements data pipeline.

# TensorFlow Input Data Pipeline

- TensorFlow implements input pipelines by creating iterators to parse the datasets.
- TensorFlow provides a thread pool to parallelize fetching and preprocessing operations.



# TensorFlow Input Data Pipeline

- TensorFlow implements pipeline by creating iterators to parse dataset.
- TensorFlow provides a thread pool to parallelize fetching and preprocessing operations.

Extract

```
dataset = tf.data.Dataset.from_tensor_slices((tf.constant(filenamees),  
                                              tf.constant(labels)))
```

Transform

```
dataset = dataset.shuffle(num_samples)  
dataset = dataset.map(parse_fn, num_parallel_calls=4)  
dataset = dataset.map(train_fn, num_parallel_calls=4)  
dataset = dataset.batch(100)  
dataset = dataset.prefetch(1)  
dataset = dataset.repeat()
```

Load

```
iterator = tf.compat.v1.data.make_initializable_iterator(dataset)  
images, labels = iterator.get_next()
```



# Presentation Outline

- ✓ *Data Pipeline*
- ✓ *TensorFlow Input Pipeline*
- *Preprocessing Steps*
- *Deep Residual Models*
- *Experimental Setup*
- *Experimental Analysis*
- *Optimizations*
- *Conclusion*

03



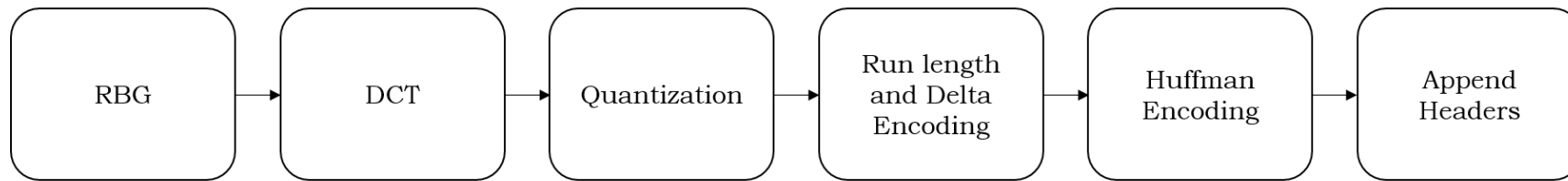
## Preprocessing

- Analysis of pre-processing steps required for training image classification models.

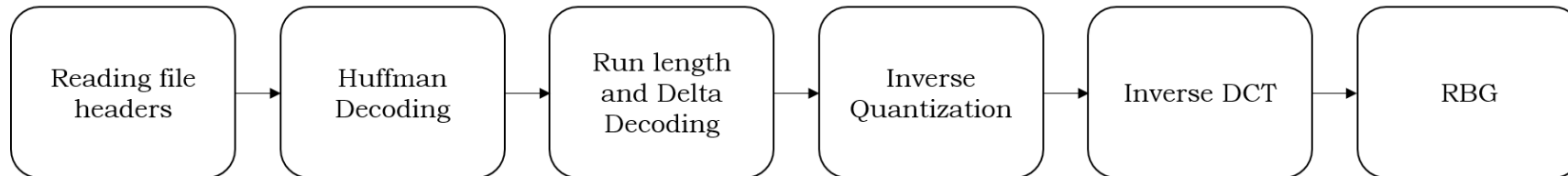


# JPEG Decoding

- JPEG is an image compression format.
- JPEG images need to be decoded (to get RGB planes) before using it in deep learning model.

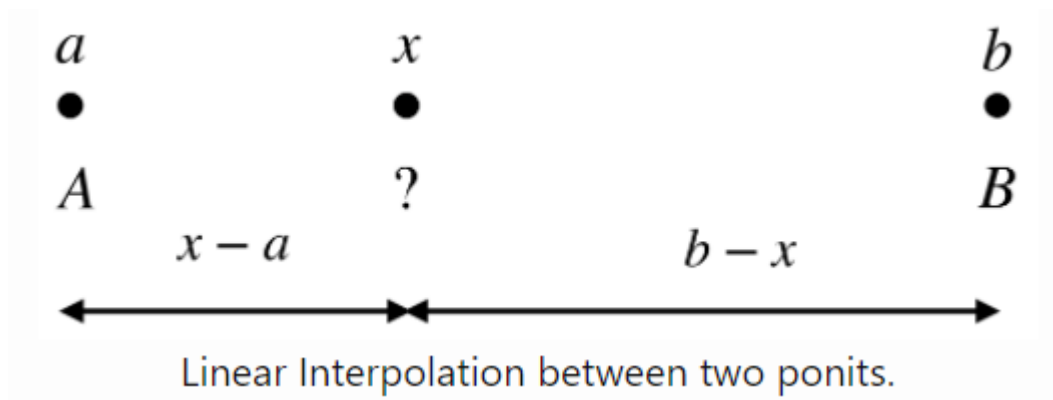


Encoding Process



Decoding Process

# Image Resizing



$$X = A \frac{b - x}{b - a} + B \frac{x - a}{b - a}$$

- TensorFlow, by default, uses Bilinear Interpolation for resizing the images.
- Linear Interpolation computes a weighted average of the values associated with the two points.
- Linear Interpolation can be transformed to Bilinear Interpolation considering linear interpolation in both x and y axis.

# Random Brightness

- Adjusting image brightness to be randomly brighter and darker
  - Required if a model may be required to perform in a variety of light settings.
  - Important to consider the maximum and minimum of brightness.

0.392	0.379	0.102	0.996
0.174	0.450	0.000	1.000
0.803	1.000	0.447	0.984
0.832	0.012	0.473	0.226

Image Pixel Matrix

+

0.125	-0.026	-0.004	-0.103
0.012	-0.114	0.043	0.105
-0.121	0.019	0.085	0.119
0.111	-0.113	0.014	-0.076

Random Brightness Matrix  
(-32/255) to (32/255)

=

0.517	0.353	0.098	0.893
0.186	0.336	0.043	1.105
0.682	1.019	0.532	1.103
0.721	-0.101	0.026	0.15

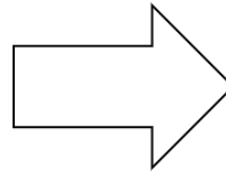
Final Image Matrix

# Image Clipping

Required to make sure that every pixel value is normalized and lies in certain range.

0.517	0.353	0.098	0.893
0.186	0.336	0.043	1.105
0.682	1.019	0.532	1.103
0.721	-0.101	0.026	0.15

Image Matrix



0.517	0.353	0.098	0.893
0.186	0.336	0.043	1.000
0.682	1.000	0.532	1.000
0.721	0.000	0.026	0.15

Clipped Image Matrix

# TensorFlow Code for Preprocessing Steps

```
def load_function(filename, label, size):  
    image_string = tf.io.read_file(filename)  
  
    image_decoded = tf.image.decode_jpeg(image_string, channels=3)  
  
    image = tf.image.convert_image_dtype(image_decoded, tf.float32)  
  
    resized_image = tf.image.resize(image, [size, size])  
  
    return resized_image, label  
  
def train_preprocess(image, label, use_random_flip):  
    if use_random_flip:  
        image = tf.image.random_flip_left_right(image)  
  
    image = tf.image.random_brightness(image, max_delta=32.0 / 255.0)  
  
    image = tf.image.random_saturation(image, lower=0.5, upper=1.5)  
  
    image = tf.clip_by_value(image, 0.0, 1.0)  
  
    return image, label
```



# Presentation Outline

- ✓ *Data Pipeline*
- ✓ *TensorFlow Input Pipeline*
- ✓ *Preprocessing Steps*
  - *Deep Residual Models*
  - *Experimental Setup*
  - *Experimental Analysis*
  - *Optimizations*
  - *Conclusion*

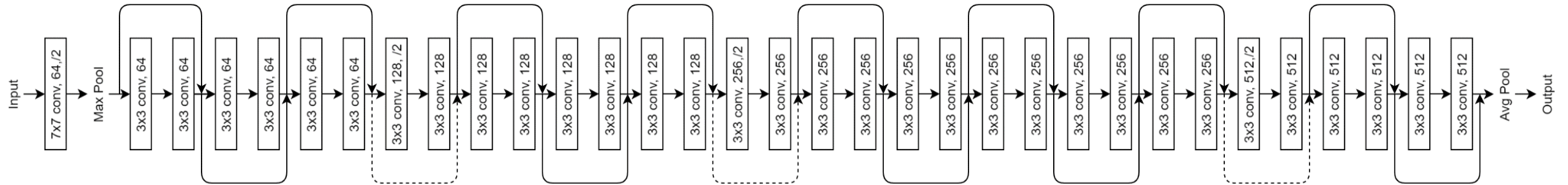
04



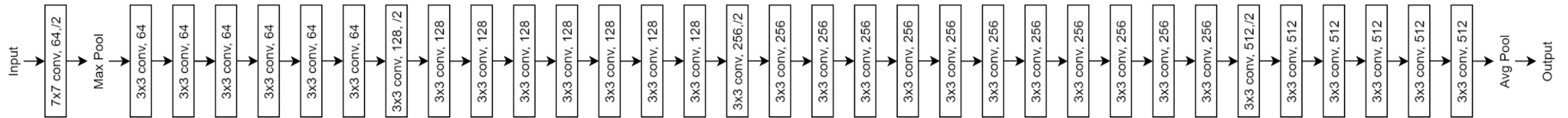
## Deep Residual Models

- Understanding Deep Residual models for image classification
- Why these models are better than CNNs

# CNN Model vs Deep Residual Model



Residual Model

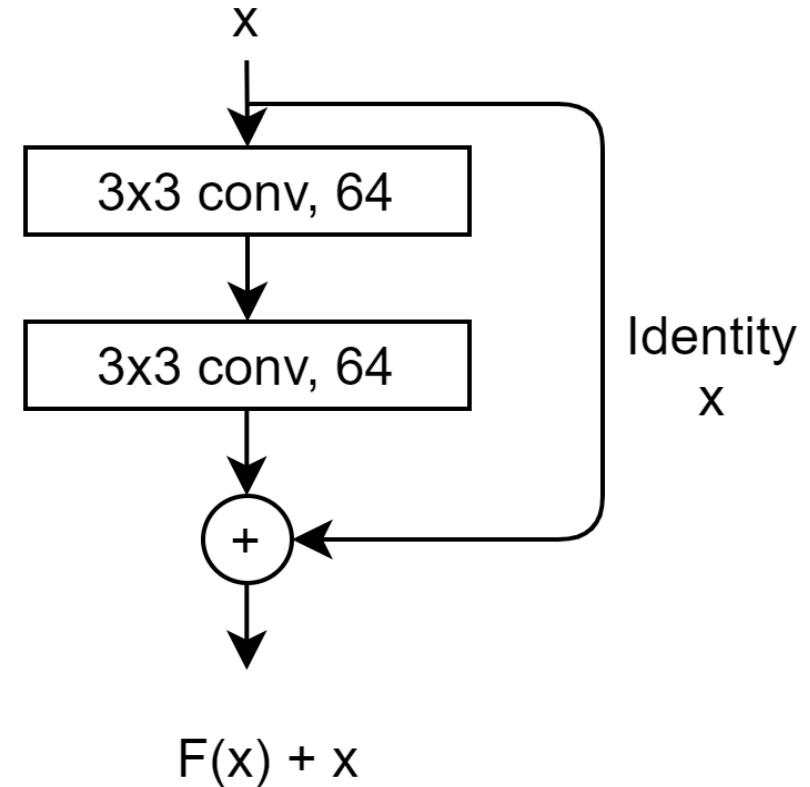


CNN Model

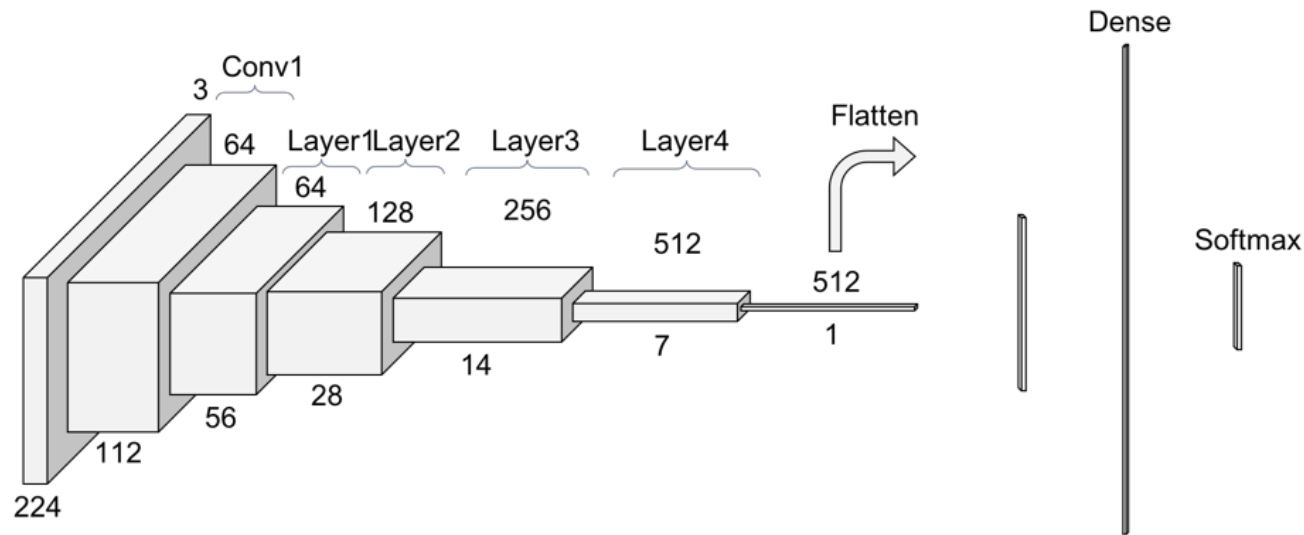


# Skip Connection

- Skip connection helps to transfer the error terms during backpropagation to the initial model layers.
- Skip connection is an identity function, which is very easy for neural networks to learn.
- Residual models are preferred over CNN model because of their ability to make the parameter updates to initial layers even in very deep models.



# Residual Model



- Each layer in Residual network decrease the dimensionality of the input vector.
- Dimensionality reduction is achieved by either kernel stride or by pooling layers.
- Each block of layers tends to learn certain number of feature sets in the input vector.

**Residual models are preferred over CNN model because Residual models are capable of learning valuable features even in deep convolutional network.**



# Presentation Outline

- ✓ *Data Pipeline*
- ✓ *TensorFlow Input Pipeline*
- ✓ *Preprocessing Steps*
- ✓ *Deep Residual Models*
  - *Experimental Setup*
  - *Experimental Analysis*
  - *Optimizations*
  - *Conclusion*

05



## Experimental Setup

- Hardware/Software used for analysis.



# Hardware

- CPU
  - Intel(R) Core(TM) i7 8564U CPU with 4 CPU cores
  - 32 GB RAM
  - 100 GB of SSD (SATA)
- GPU
  - Nvidia Tesla V100-SXM2-16GB
  - 16 GB global memory, supports both L1 (128 KB for each SM) and L2 caches (6 MB)
  - 5120 CUDA cores and supports CUDA 7.0 (with OpenCL 1.2)

# Software

- TensorFlow 2.4.1
  - CUDA® 11.0 GPU toolkit
  - cuDNN 8.0.4
  - Tensorboard 2.0
- ImageNet Dataset
  - 14,197,122 unique images
  - 2000 unique image classes
  - Raw images in JPEG format





# Presentation Outline

- ✓ *Data Pipeline*
- ✓ *TensorFlow Input Pipeline*
- ✓ *Preprocessing Steps*
- ✓ *Deep Residual Models*
- ✓ *Experimental Setup*
- *Experimental Analysis*
- *Optimizations*
- *Conclusion*

05

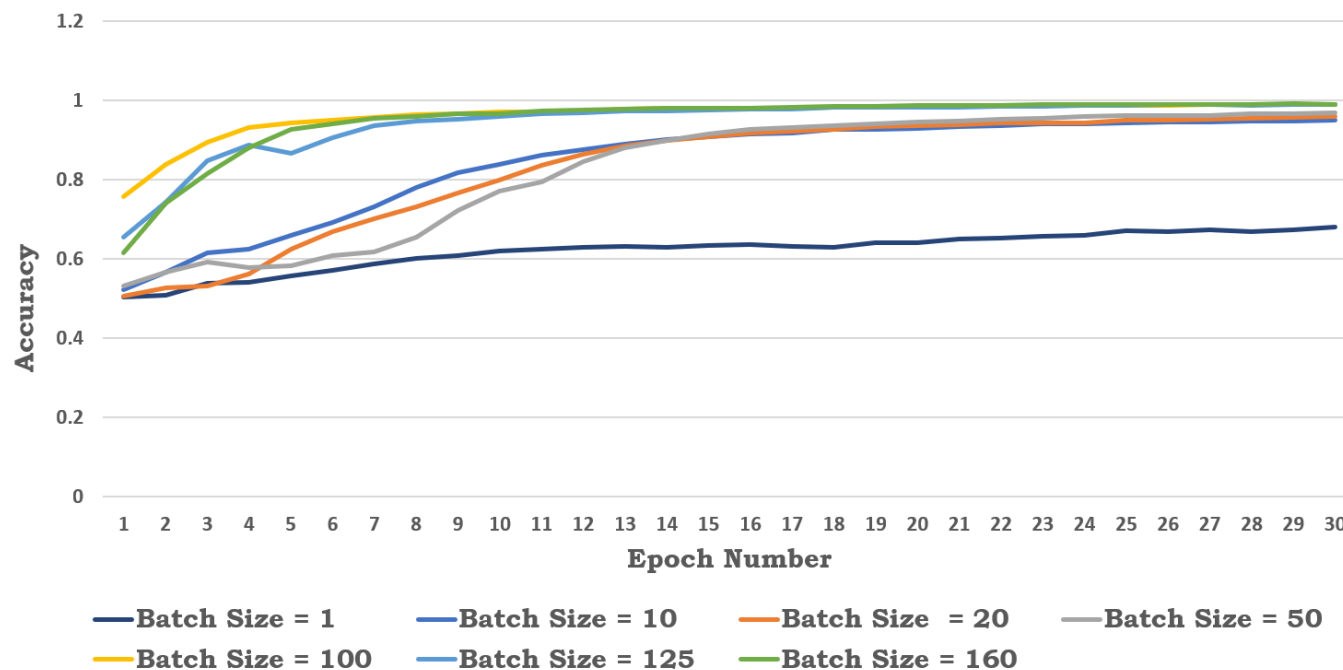


## Experimental Analysis of Data Pipeline

- Performance of TensorFlow models and pipelines.

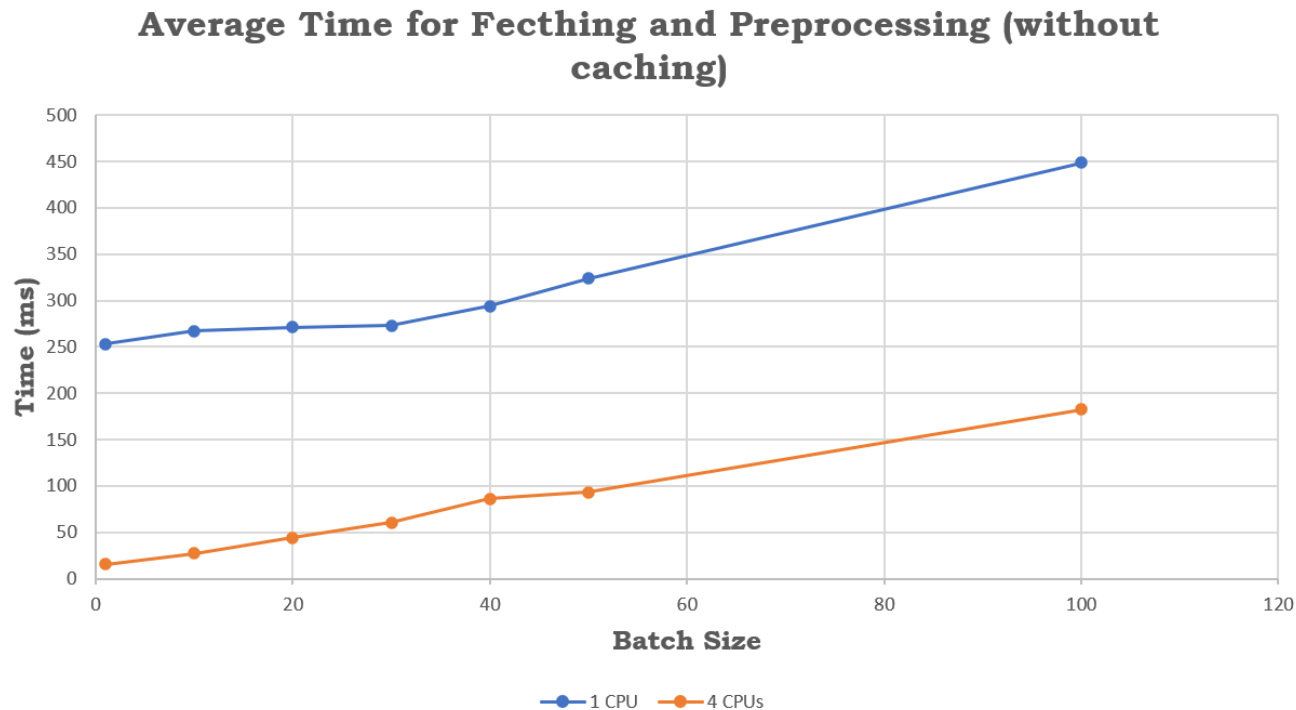


# Resnet50 Model Accuracy



- Smaller batch size doesn't achieve high accuracy, for instance batch size equal to 1. This is because of overfitting.
- As batch size is increased, the accuracy of model increases.
- As the batch size increases, the higher accuracy is achieved in lesser number of epochs.
- However, the batch size supported by GPU depends upon factors such as global GPU memory, size of the training model.

# Average Fetching and Preprocessing Time

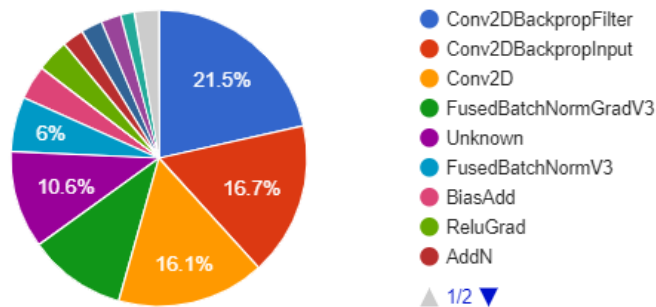


- For this measurement, page caches are flushed to make sure no raw data is cached.
- Graph shows significant improvement in fetching and preprocessing latencies with more number of I/O worker threads.

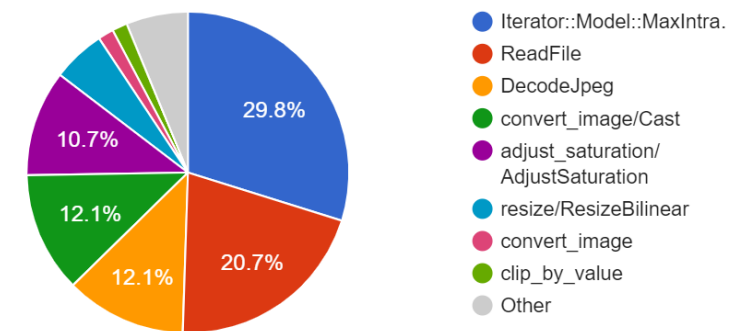
**This indicates that CPU is a bottleneck, when CPU is doing both fetching and preprocessing operations.**

# Breakdown of CPU/GPU time

ON DEVICE: TOTAL SELF-TIME (GROUPED BY TYPE)  
(in microseconds) of a TensorFlow operation type



ON HOST: TOTAL SELF-TIME (GROUPED BY TYPE)  
(in microseconds) of a TensorFlow operation type



- ~20% of the average total time is spent in directly accessing an image from SSD.
- Rest ~80% of time is spent for shuffling data, preprocessing data and creating batches.

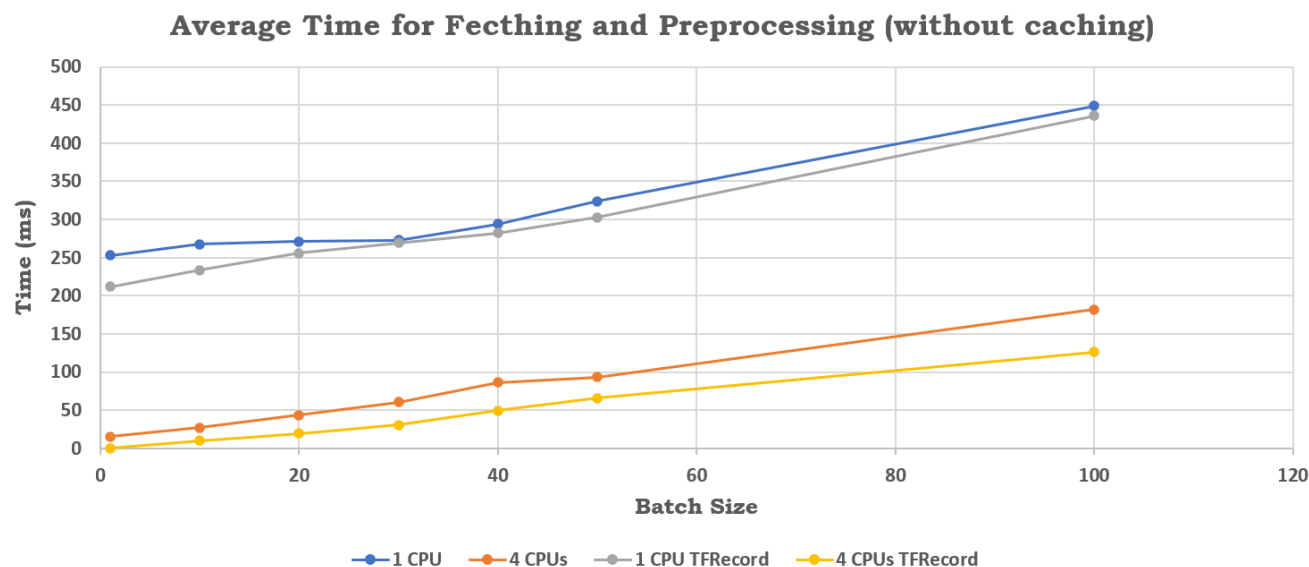
**~50% of CPU time is spent on preprocessing operations such as *DecodeJpeg*, *ResizeBilinear*, *AdjustSaturation* etc.**



# Data Compression

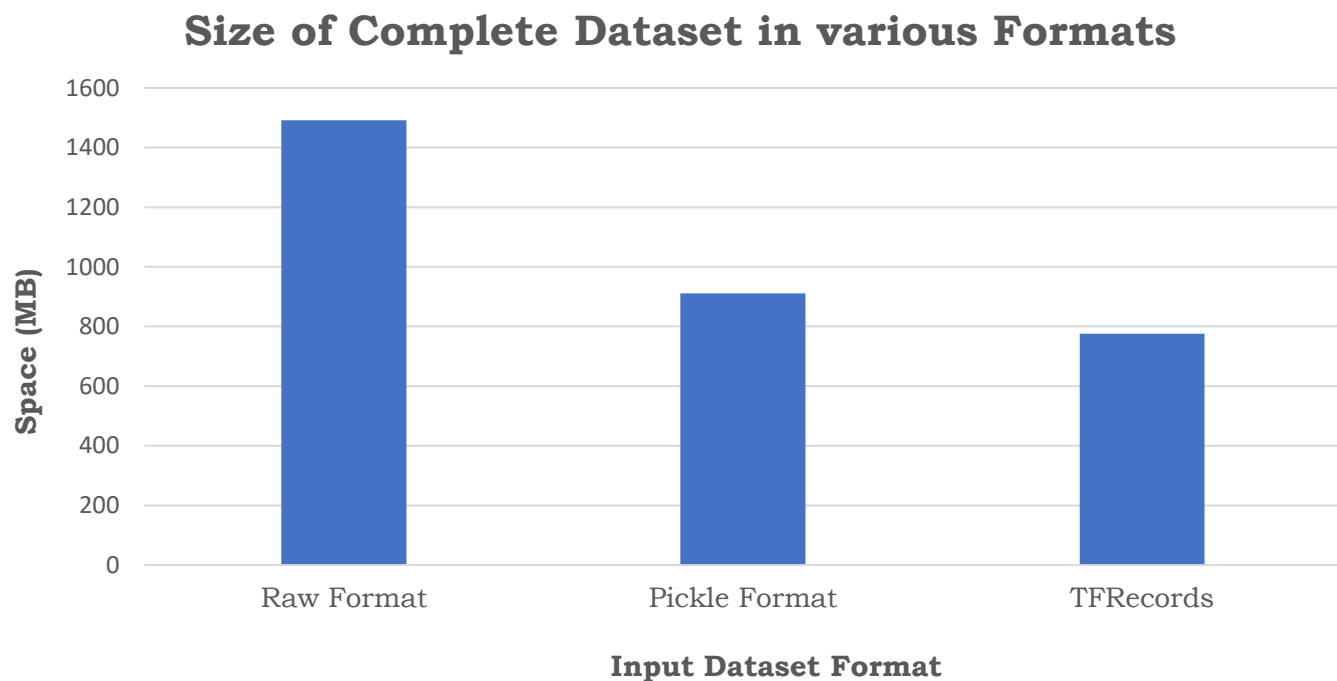
- TFRecords
  - Consume less space on disk
  - Efficient read/write operations
  - TensorFlow use *snappy* and *zlib* libraries
  - Dynamic formats for compression and decompression
- Python Pickling
  - Doesn't support dynamic formats
  - Doesn't work well with data sharing between other language platforms.

# Performance of Data Pipeline with Compressed Data



- Using compressed TFRecords in place of raw input images, the average time for fetching and preprocessing decreases.
- However, the difference is not significant as compared to the average latencies in case of raw input images.

# Memory Requirement for various Dataset formats



- Using TFRecord format, ~45% disk space is saved.
- Using Python Pickle format, ~28% of disk space is saved.

**Compressed dataset saves a lot of space on NV Memory, however its impact on the overall model performance is very small.**



# Presentation Outline

- ✓ *Data Pipeline*
- ✓ *TensorFlow Input Pipeline*
- ✓ *Preprocessing Steps*
- ✓ *Deep Residual Models*
- ✓ *Experimental Setup*
- ✓ *Experimental Analysis*
- *Optimizations*
- *Conclusion*

06

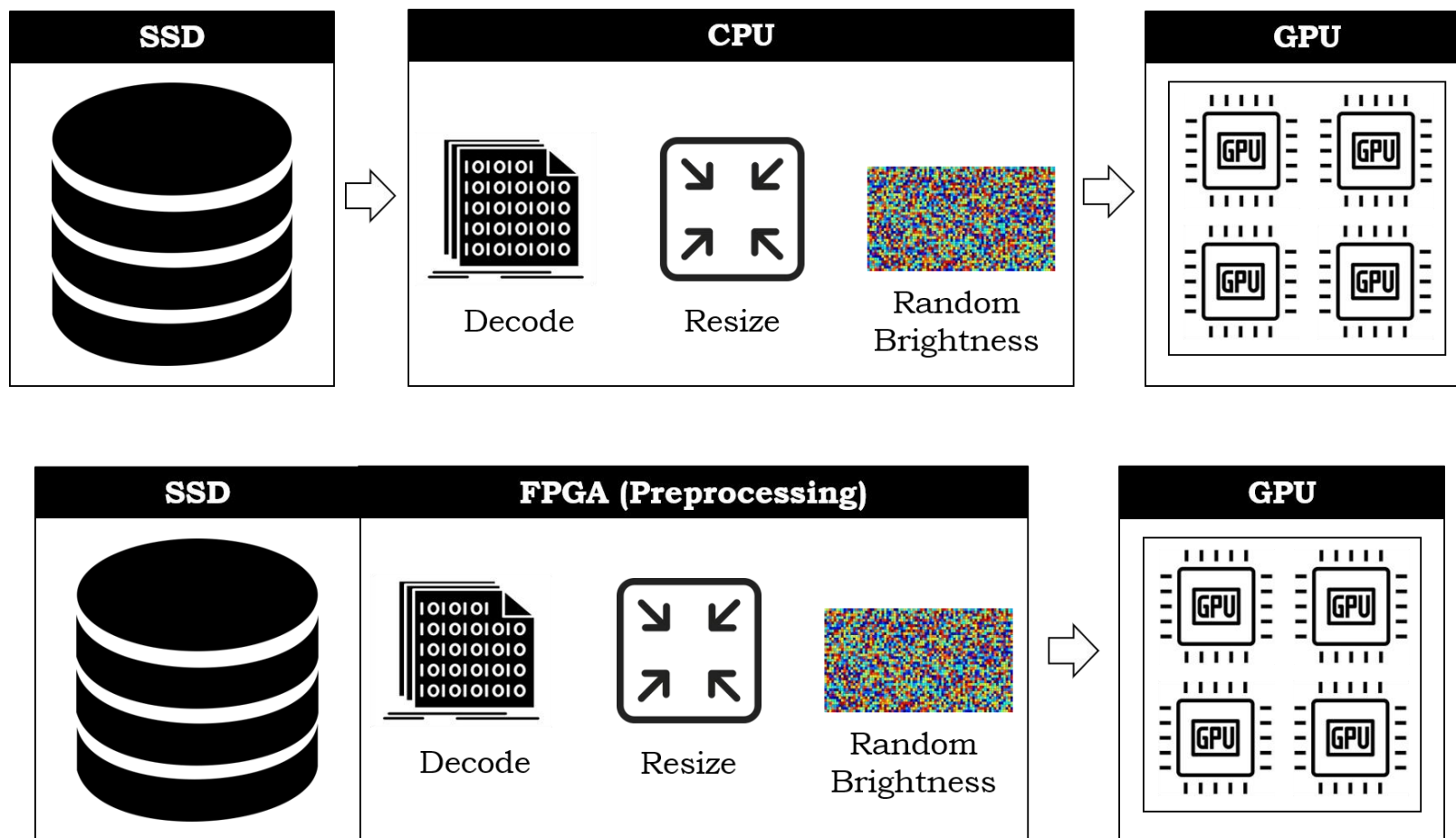


## Optimizations

- How the preprocessing can be realistically moved to SSDs?
- Ways to mitigate various bottlenecks in the pipeline design.



# Moving the preprocessing steps to SmartSSD FPGA



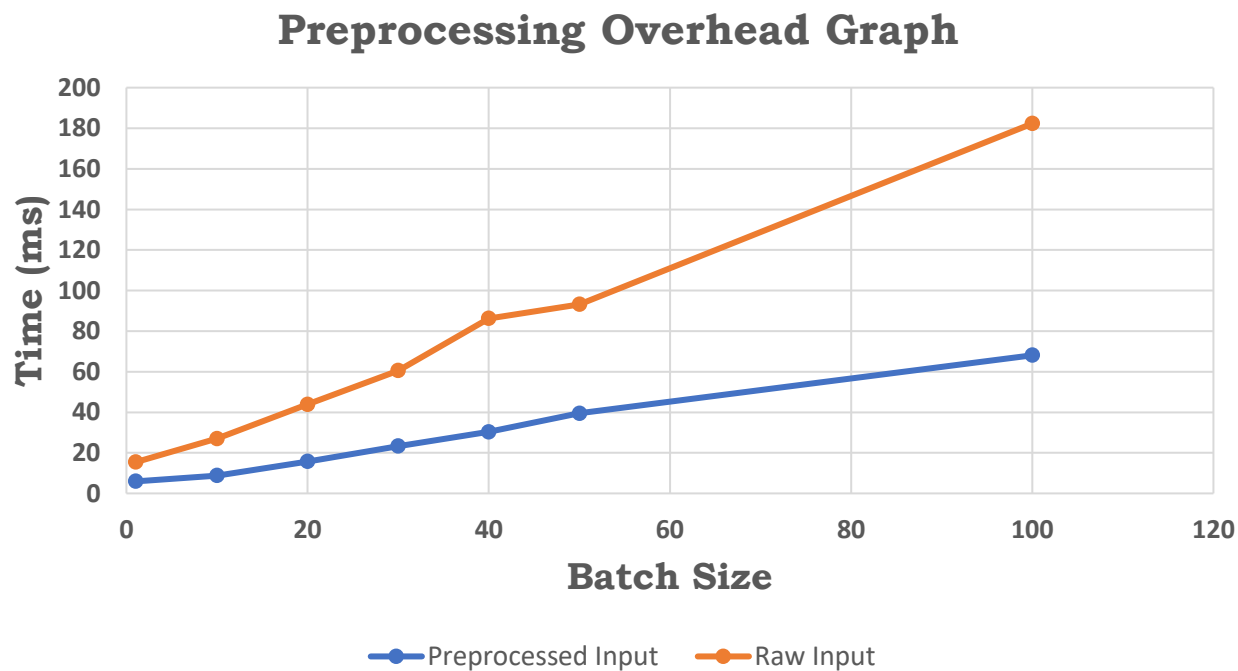


---

## **Following preprocessing steps can be moved to SmartSSD FPGA:**

- Pico JPEG Decoder – Implementation by Intel using OpenCL.
- Bilinear Interpolation – Implemented to extract RGB planes individually and resize each plane.
- Adding random brightness – Implemented using a pseudo-random number generator provided by *stdlib*.
- Normalization and Clipping – Implemented to convert integer pixel values to float type values between 0 and 1.

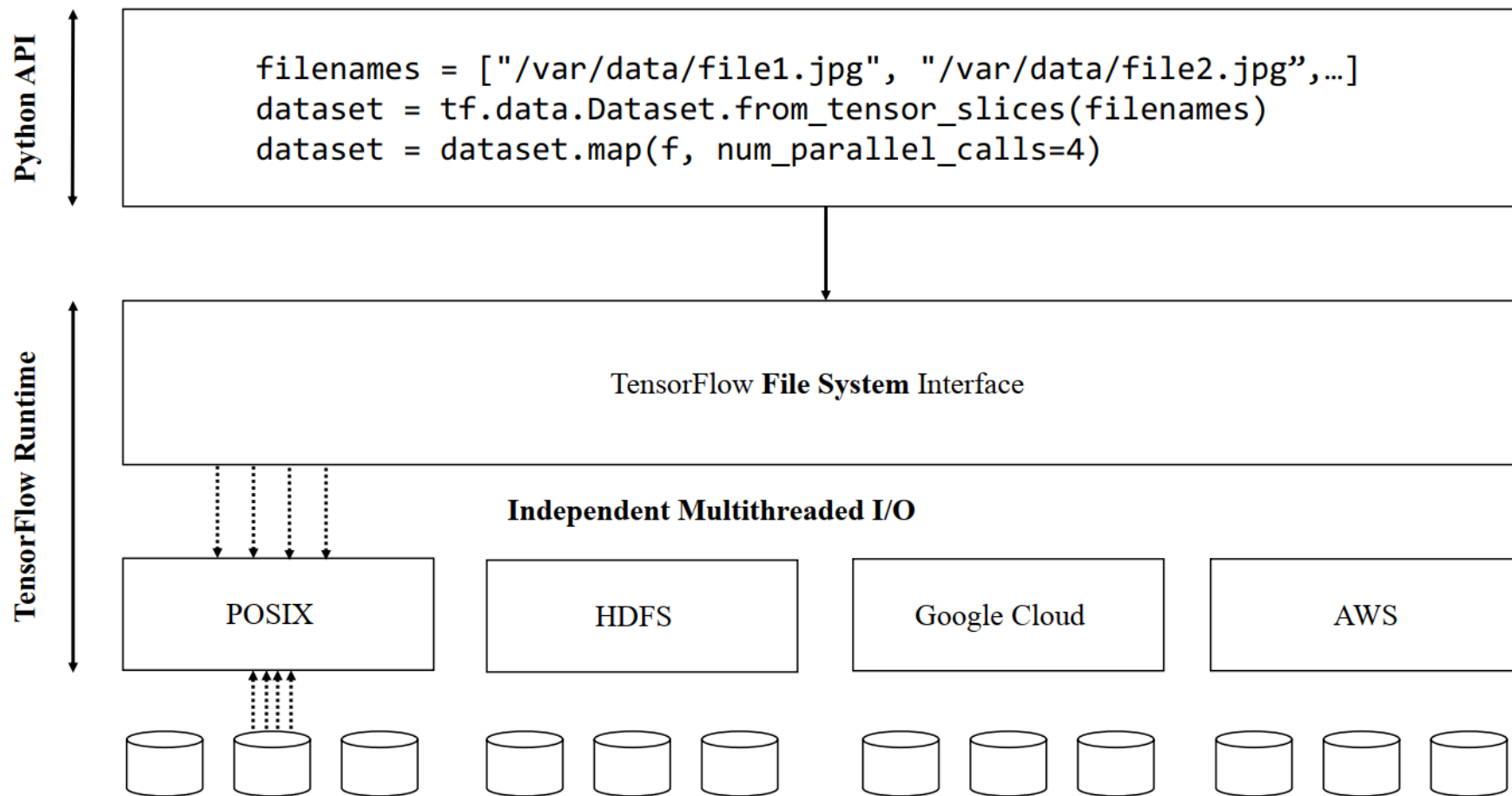
# Average Time for Preprocessed and Raw Input Images



- Data is preprocessed offline.
- Data pipeline is fed with preprocessed data instead of raw JPEG images.
- Graph shows the CPU execution overhead caused by the preprocessing steps in input pipeline.

**Significant improvement in performance of input pipeline as the preprocessing steps are offloaded.**

# Custom Filesystem Integration



# TensorFlow Filesystem Class

```
class Filesystem {
// File creation
virtual Status NewRandomAccessFile(const string& fname, std::unique_ptr<RandomAccessFile>* result) = 0;
virtual Status NewWritableFile(const string& fname, std::unique_ptr<WritableFile>* result) = 0;
virtual Status NewAppendableFile(const string& fname, std::unique_ptr<WritableFile>* result) = 0;
virtual Status NewReadOnlyMemoryRegionFromFile(const string& fname, std::unique_ptr<ReadOnlyMemoryRegionFile>* result) = 0;

// Creating directories
virtual Status CreateDir(const string& dirname) = 0;
virtual Status RecursivelyCreateDir(const string& dirname);

// Deleting
virtual Status DeleteFile(const string& fname) = 0;
virtual Status DeleteDir(const string& dirname) = 0;
virtual Status DeleteRecursively(const string& dirname, int64* undeleted_files, int64* undeleted_dirs);

// Changing directory contents
virtual Status RenameFile(const string& src, const string& target) = 0;
virtual Status CopyFile(const string& src, const string& target);

// Filesystem information
virtual Status FileExists(const string& fname) = 0;
virtual bool FilesExist(const std::vector<string>& files, std::vector<Status>* status);
virtual Status GetChildren(const string& dir, std::vector<string>* result) = 0;
virtual Status Stat(const string& fname, FileStatistics* stat) = 0;
virtual Status IsDirectory(const string& fname);
virtual Status GetFileSize(const string& fname, uint64* file_size) = 0;

// Globbing
virtual Status GetMatchingPaths(const string& pattern, std::vector<string>* results) = 0;

// Misc
virtual void FlushCaches();
virtual string TranslateName(const string& name) const;
};
```

[https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/platform/file\\_system.h](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/platform/file_system.h)

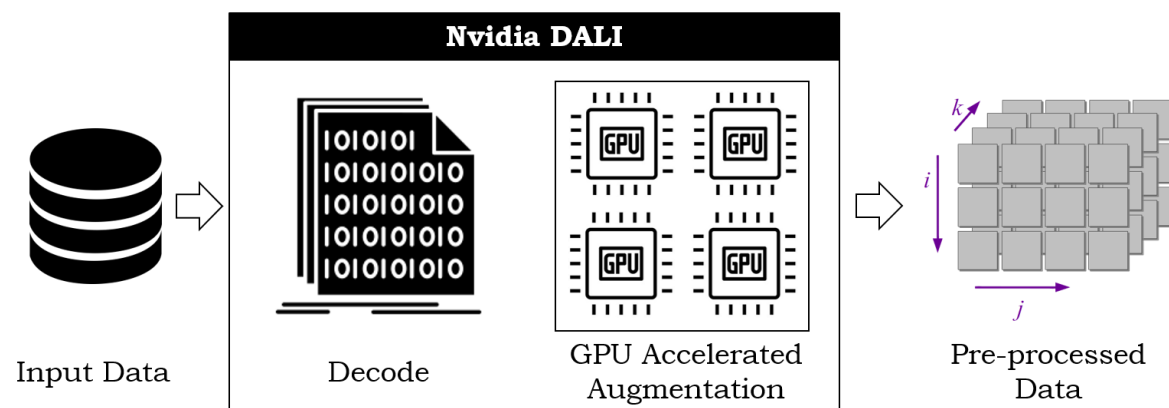
- TensorFlow *FileSystem* interface as a subclass.
- Implement subclasses
  - *RandomAccessFile*
  - *WritableFile*
  - *AppendableFile*
  - *ReadOnlyMemoryRegion*
- Register the *FileSystem* implementation with an appropriate prefix pattern.

## TensorFlow Filesystem Class

```
tf.io.gfile.GFile("/path/to/file")
tf.io.gfile.GFile("file:///path/to/file")
tf.io.gfile.GFile("gs:///path/to/file")
tf.io.gfile.GFile("s3:///path/to/file")
tf.io.gfile.GFile("hdfs:///path/to/file")
tf.io.gfile.GFile("viewfs:///path/to/file")
tf.io.gfile.GFile("https://my.awesome.site/path/to/file")
#  URI scheme:      |<---->|
#  URI host:        |<----->|
#  URI path:        |<----->|
```

**Files are identified by URI (Uniform Resource Identifier)**  
[< *scheme* >: //[< *host* >]] < *filename* >

# Nvidia DALI



Nvidia DALI Preprocessing

- Provides Python APIs.
- Supports LMDB, RecordIO, TFRecord, COCO, JPEG, JPEG 2000, WAV, FLAC, OGG, H.264, VP9 and HEVC.
- Supports CPU and GPU execution.
- Allows direct data path between storage and GPU memory with GPUDirect Storage.



# Presentation Outline

- ✓ *Data Pipeline*
- ✓ *TensorFlow Input Pipeline*
- ✓ *Preprocessing Steps*
- ✓ *Deep Residual Models*
- ✓ *Experimental Setup*
- ✓ *Experimental Analysis*
- ✓ *Optimizations*
- *Conclusion*





06



## Conclusion

- Summary of outcomes of the project.

- 
- Analyzed various stages of end-to-end machine learning model to understand
    - Input data pipeline and various stalls associated with it.
    - Resnet model and how its trained.
    - Performance of various stages of input data pipeline and identifying the bottlenecks.
  - Studied the feasibility to ensure the integration of SmartSSD solution with existing Deep Learning frameworks
    - By understanding the provision of loading the custom filesystem with TensorFlow runtime.
    - By understanding the individual steps executed by DL frameworks for preprocessing image datasets.

- 
- Implemented preliminary code for FPGA implementation following preprocessing steps:
    - Decode JPEG image
    - Resize RGB image planes
    - Normalization and adding random brightness
    - Clipping

This project presents a preliminary analysis of a potential solution to reduce the CPU bottlenecks in future by using SmartSSD solutions. This is essential for the performance of deep learning models as GPU are getting more and more efficient, however CPU efficiencies are almost stagnant for over a decade.



**Thank You**

