Input Data Pipeline Analysis of TensorFlow Models

FNU Sachin
ECE
Virginia Tech
Virginia, USA
sachin255701@vt.edu

Dr. Changwoo Min
ECE
Virginia Tech
Virginia, USA
changwoo@vt.edu

***Abstract*— Training of machine learning models is a data and resource intensive process. A large amount of research is being done in optimization of ML models to get better accuracy and lower training times by adopting various optimizations for GPU memory utilizations, reducing CPU-GPU communication overheads and compiler optimizations. However, the impact of storage systems on the training time of the ML models is a rather unexplored front in the machine learning domain. As deep learning techniques are becoming more efficient from the computational front, the focus must also be shifted towards how the storage media are becoming more and more efficient and how to leverage these advancements to reduce I/O bottlenecks in models. The latest advancement in storage domains, such as SmartSSDs are equipped with high end FPGAs for increasing the efficiency of data intensive applications. So, this project mainly focuses on deeply understanding the basic architecture of input data pipeline in TensorFlow deep learning framework and analyzing potential bottlenecks in the data path. The project will present the thorough analysis and feasibility of offloading the data intensive parts of the pipeline to FPGA inside SmartSSD. The project presents the analysis of input pipeline by training Resnet50 model, used for image classification, with ImageNet dataset and feasibility and impact of moving pre-processing step to SmartSSD.**

Keywords—Deep Learning, Storage, FPGA, Computer Arechitecture.

I. INTRODUCTION

Data is the essence of machine learning and deep learning models. The state-of-the-art accuracy of deep learning models can be attributed to the large training datasets. Deep Neural Networks (DNN) have proved themselves to be very efficient in solving problems such as image recognition, voice recognition etc. However, training these DNNs is a very resource and data intensive process. The DNN training process requires the various resources such as Disks to store input data, CPU for fetching and pre-processing the input data and GPUs for performing complex computations to reach an optimized solution for the minimum of the cost function. So, each training step involves the coordination of various resources, however the main stream research [1] for the optimization of DNNs is mainly focused on optimization of GPU memory usage, communication between CPUs and GPUs or other compiler-based optimizations. However, the impact of storage media and the pre-processing pipeline is rather unexplored in case of deep learning models. Understanding of storage media and pr-processing pipeline is a crucial step in analysis of their impact on overall performance of the model.

Main motivation behind this process comes from the advent of SmartSSDs, which are the storage media that are equipped with FPGAs for acceleration of data intensive applications. The Samsung SmartSSD is one of the kind Computational Storage Device (CSD) which has integrated Xilinx FPGA platform that can be used to accelerate the I/O operations. The FPGA inside opens up numerous possibilities in which the FPGA can be used for optimization of machine learning models. One such possibility, and the main focus of our project, is moving the data pre-processing pipeline to the FPGA and reducing the CPU load. This project will mainly focus on understanding the whole input data pipeline for training deep learning models and analysis of feasibility of moving the pre-processing pipeline or at least some part of it to the FPGA inside SSD.

II. INPUT DATA PIPELINE

During the training of ML models, the training process runs through various epochs, where in each epoch of training the dataset is passed through ETL (Extract, Transform and Load) phase. In ETL paradigm has following three phases, as shown in Fig 1.:

1. The *Extract* subphase is fetching the data from storage. In this project, our complete dataset is fetched from an SSD connected locally to the system.

2. The *Transform* subphase is to pre-process the data for training the model. This phase includes decoding of images, transforming them to desired size supported by ML models etc. This process is also responsible for collating the images into batches for the pre-processing steps.

3. The *Load* subphase is providing the data to compute clusters for actual training of models. In this step, data is copied from local CPU memory to GPU global memory and initiating GPU kernel to process data.

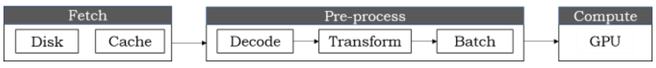


Fig. 1. Input Data Pipeline

A. *Sequential Data Model versus Pipelined Data Model*

The traditional approaches, used by machine learning frameworks, adopt the sequential model for implementing ETL phase during epoch training. The sequential model relies on the fact that the CPU first performs all the fetching and pre-processing operations to the whole dataset prior to starting any training operation. In this type of sequential model, the GPU resources are wasted as when the CPU is performing fetching and pre-processing operations in the entire dataset, GPU compute has no data to work upon. This is clearly illustrated in Fig. 2. We can see that once the CPU is done with pre-processing work, it copies data to GPU memory and the GPU starts executing the kernel. Once GPU kernel operations are done, the resultant data is copied back to CPU local memory.

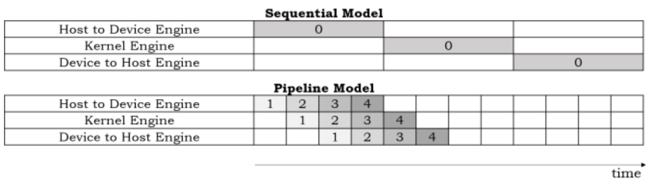


Fig. 2. Comparison between Sequential model versus Pipelined model

This type of sequential approach wastes useful resources and causes bottlenecks in the system. One other limitation with the sequential approach is that the typical sizes of input dataset are very large these days and even modern systems don't have enough resources to fetch and pre-process the whole dataset in one iteration. So, the dataset is broken up into the smaller batches that can fit in CPU and GPU memories. But even using batches of dataset, the sequential model is not efficient as resources are not utilized efficiently during the training process.

The alternative approach that is used by the ML frameworks, such as TensorFlow and PyTorch, is to create a pipeline, by overlapping the execution of various resources in parallel. In this scenario, instead of using the whole dataset in one iteration, the dataset is split up into various small batches. Then starting with an individual batch, that batch is being fetched and pre-processed by the CPU. Once done, then this batch is fed to the GPU for model training operation. Meanwhile CPU is free again to fetch and pre-process the next batch. This way GPU can be kept busy while the CPU is prefetching the next batch from disk. This way the ML framework is able to utilize the available resources efficiently and parallelly. So, technically the input data pipeline can operate in parallel to GPUs i.e., input data pipeline is capable of providing the pre-processed data to GPU, to utilize them efficiently. This is illustrated in the timeline shown in Fig 2.

B. *Input Data Pipeline using TensorFlow Dataset API*

TensorFlow is an open-source software library provided by Google that is used for a large variety of tasks related to machine and deep learning domains. TensorFlow has C++ runtime interface and provides Python wrappers around the runtime APIs for the ease of use. The TensorFlow runtime is responsible for efficiently handling I/O operations, managing data movement between host and device, memory allocations etc. In this project, our main focus is to study how TensorFlow handles the I/O operations during model training. As mentioned in the last section, the TensorFlow provides a mechanism to create an efficient input pipeline, which emulates the pipelined behavior as a producer consumer model. The input pipeline (producer) is responsible for providing data to GPU compute (consumer). As the GPU (consumer) works on the training the model, the input pipeline parallelly prepares the next batch of data for the GPU to work on. Essentially the TensorFlow enables input pipeline processing in parallel to model computation in GPU, which optimizes the overall performance. By overlapping the preparation of batches with the computation, new batches can be prefetched and become ready for processing as soon as the computation pipeline becomes available. The input pipeline is kept busy fetching the next batch as soon as the current batch is fed to the GPU, rather than waiting for the computation pipeline to request for a new one. Since, the input pipeline operates on CPU and computation is typically done on GPU, this prevents GPU from starving for the input data.

For enabling this feature, TensorFlow provides *tf.data* API that capable of performing the following operations:

1. Fetching data from storage modules. This API also enables the user to aggregate data from different nodes in a distributed file system.
2. API is capable of applying various transformations to input dataset.
3. Merging of various input data points from a dataset into mini-batches, that are used for compute processing.

The TensorFlow uses *tf.data.Dataset* class to generate an efficient input pipeline. This class provides the mechanism to generate the iterator for input dataset, which is used for parallelly fetching the elements of the dataset. For this project, as we are working with a dataset that contains raw images, we have to create a data pipeline that is capable of reading raw image files from disk, decoding those image files to get RGB arrays, transforming the images to desired sizes, create baches from random set of images. The input data pipeline, that is created for this project, can be majorly performing following operations to process raw input images from the dataset:

1. *I/O Operation:* The I/O operations for this project includes reading of raw image files. For the purpose of this project, we have various raw JPEG files as our input dataset. TensorFlow has the capability to generate an iterator that provides the file names present in the dataset. The API *from_tensor_slices* is used to create an iterator, from list of absolute paths of all the files in dataset, that can fetch the names of individual image files from the train and test datasets, apply the required transformations and then collate them into the batches.

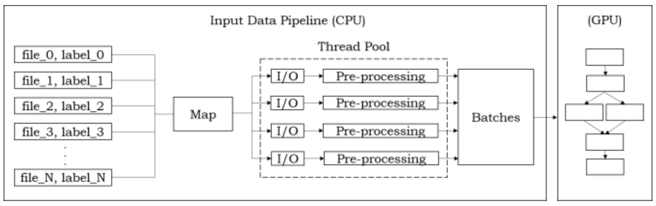


Fig. 3. Internal of Input Data Pipeline

Since the input files are independent from one another, this whole process can be highly parallelized. TensorFlow achieves this parallelism by providing *num_parallel_calls* argument, which specifies the number of available CPU cores that can be used to parallelly fetch one element of a dataset and preprocess it. Apart from above following operations, the input pipeline also provides APIs such as *batche(size)* to specify batch size to be used during training, *prefetch(num_batches)* to prefetch specified number of batches and *shuffle()* to shuffle the inputs to form a batch from random elements from input dataset.

2. *Pre-processing:* After reading each file from a storage module, the input pipeline also provides a mechanism to apply transformations to the fetched images. For this purpose, TensorFlow provides a map API. This API accepts

the name of transformation functions as an argument. For the purpose of this project, we mainly apply transformations such as decoding, resizing and adding random brightness, which will be explained in the next section in detail.

For decoding JPEG images and obtaining the RGB format data, TensorFlow provides an API *tf.image.decode_jpeg*, which performs the decoding operation using one image at a time and returning the decoded image in RGB format. The decoded image is converted into a tensor using API *tf.image.convert_image_dtype*. Once the RGB tensor is ready, the decoded images can be resized using API *tf.image.resize* to a specific size that is used by the input layer of the machine learning network. After this one more transformation is applied on the decoded images to add random brightness to them using API *tf.image.random_brightness*. This transformation is useful for the models that are required to do accurate classification in different light settings. The last transformation is clipping of the pixel values to make sure all the pixel values are in specified range which helps in preventing exploding gradients problem with sensitive models.

3. *Checkpointing:* Checkpoints are the hooks in the TensorFlow code that are responsible to capture the exact values of all the training parameters inside the model. TensorFlow provides an API *tf.train.Saver()* to create a checkpoint, at which point all the parameters are saved to disk. This particular feature is very useful to restart the training computations of the model from a previous checkpoint. While saving checkpoint data, TensorFlow mainly generates three files, first one being the metadata file which stores information about structure of graph, second one being the index file that stores information about the tensor/ variables being used and the last one being the data file that store the actual values of variables/tensors. These files have *.meta*, *.index* and *.data* extensions respectively. Whenever a checkpoint is restored, TensorFlow first restores the computation graph from the metadata file and then restores the variables and their corresponding values.

For this project, these are the only features of input data pipelines that are studied thoroughly. In the next section, we will present the experimental deep learning model that we have used to test the input data pipeline and measure the performance of each step.

III. EXPERIMENTAL MODEL

Creating an efficient deep learning model has its own problems associated with it. The researchers have tried to make deep learning models deeper and efficient by adding extra hidden layers in the model to enable it to learn various hidden features during the training steps. But adding an extra layer has not always resulted in increasing the accuracy of the deep learning models. Due to problems such as vanishing and exploding gradients, the deeper neural networks are difficult to train. The experimental research has proved that as we increase the depth of the neural network, the accuracy starts to saturate and then degrade as we run through the training process. The research paper [2] postulates that the degradation in accuracy is not actually caused by the overfitting of models. However, the degradation is mainly caused due to vanishing gradients problem, that causes gradient changes, not to reach initial layers, during the back propagation step of training. Because of this problem, the parameters in initial layers don't get updated efficiently, hence saturating the accuracy of the model.

For this project, we use Resnet50 model [2], which is a residual deep learning model used for classification of images in various categories to produce corresponding labels. Fig 4 shows the high-level structure of the Resnet50 architecture. Resent50 architecture consists of 5 main blocks, each with several Convolutional layers. As Fig. 4 shows, these five blocks are labelled as *Conv1, Layer1, Layer2, Layer3, Layer4.* This block diagram clearly shows how the dimensions of input raw image (224 X 244 X 3) are reduced in each block, eventually producing a 1-D matrix to get the probabilities of the classification in one particular class. All blocks shown in Fig 4 consist of basic Convolution layers, which is also represented as *conv* in Fig. 6 (which is detailed layer wise design of Resnet50 Model). Each convolution layer mainly performs three major tasks, which are state as follows:

1. Convolution Operation: In this operation, the input volume is convoluted with kernel/filters. Kernels/Filters are actually used for extraction of different features (also known as feature map) inside the input image. Before forming a convolutional layer, the kernel size must be mentioned as the kernel size remains fixed for each layer. For Resnet50 Model, initial convolutional layer i.e., *conv1* uses a kernel of 7x7, rest all the convolution layers use a kernel of size 3x3.

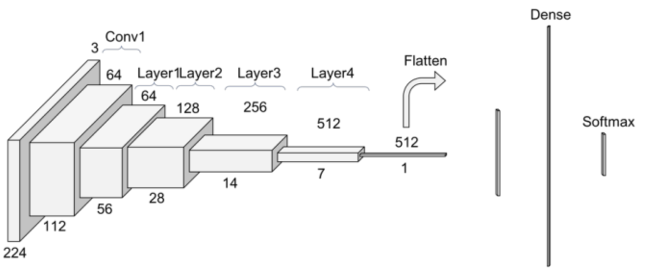


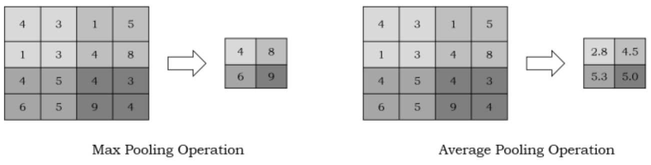
Fig. 4. High level architecture of Resnet50 model

2. Batch Normalization operation: This operation is used for re-scaling the coefficients of the feature map learned by the layers. Batch normalization scales layers outputs to have mean 0 and variance 1. The outputs are scaled in such a way to train the network faster. It also reduces problems due to poor parameter initialization. For Resnet50 architecture, the Z-normalization is used on batch operations.

3. Activation operation: As all of the above steps are linear in nature, the activation functions are used to introduce non-linearity in the model. For the Resnet50 model, ReLU (Rectified Linear Unit) activation function is used, which outputs the input directly if its positive otherwise outputs 0.

Apart from the convolutional layer, the Resnet50 Model is also required to reduce the dimensionality of input volume for each block. The main purpose of dimensionality reduction in each block is to reduce the total number of training parameters in each block. If no dimensionality reduction is done, the training process will become very slow as there are lots of parameters that must be learned. There are two ways employed by the Resnet50 Model to achieve dimensionality reduction states as follow:

1. Pooling Operation: Pooling layer is responsible for reducing or down sampling of input volume. There are two types of Pooling Layers used in the Resent50 Model, one is Max Pooling Layer and another one is Average Pooling Layer. Pooling layers are only used at start and end of the whole model. The functionality of Max and Average Pooling Layer is shown in figure below.



2. Kernel Stride: Kernel stride is another way of reducing dimensionality which is used by the layers inside the block. This method of down sampling is used in every block of the Resnet50 model. When computing the convolution operation, we start with the convolution window at the top-left corner of the input tensor, and then slide it over all locations both down and to the right. In the Resnet50 model, the down sampling is achieved by keeping the kernel stride equal to one.

Block diagrams in Fig 5 shows comparison of a simple CNN model with 50 layers on the left side and Resnet model with 50 layers on the right side. The main difference between these two models is the presence of skip connection is Resnet50. Skip connection, shown in Fig. 5, is used to ease the flow of gradients during the back propagation step of training i.e., it helps in eliminating the vanishing gradient problem by learning the identity function.

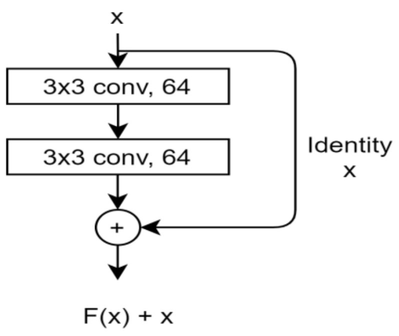


Fig. 5. Skip Connection in Resnet50 model

Dotted connections shown in Fig. 6 are also skip connections, but not with the identity function but with the convolution function that is used to change the dimensionality of the data. This is required because whenever the data flows from one block to another its dimensionality is reduced, so using the same dimensionality by identity skip connection will not work. So, the input data is convoluted with 1x1 kernel with kernel stride of one to reshape the data to match the dimensions used in the block. Another type of skip connection is called Bottleneck Layer, that uses a stack of 3 layers instead of 2. The three layers are 1×1, 3×3, and 1×1 convolutions, where the 1×1 layers are responsible for reducing and then increasing (restoring) dimensions, leaving the 3×3 layer a bottleneck with smaller input/output dimensions.

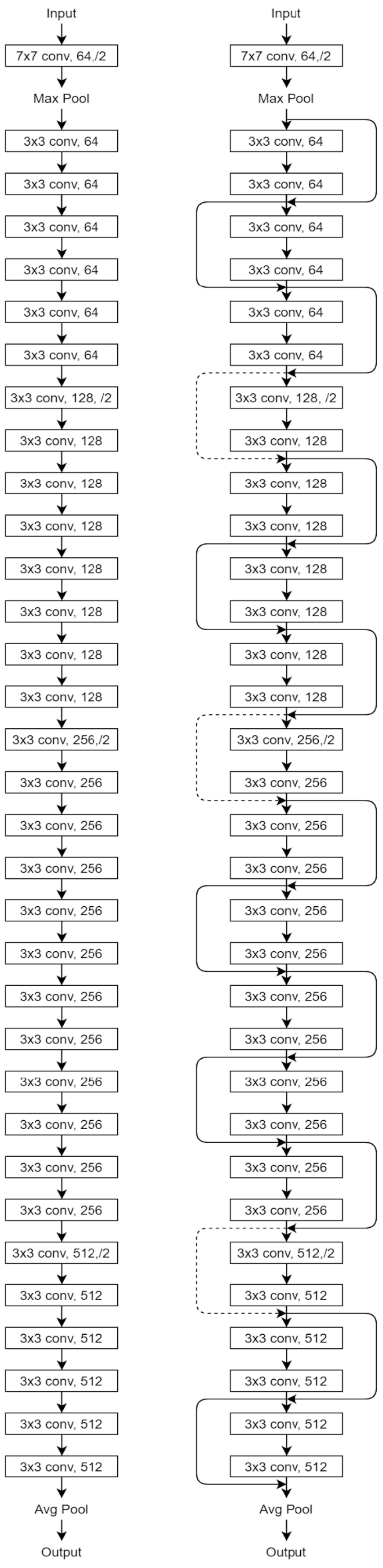


Fig. 6. Simple CNN Model (Left) va Resnet Model (Right)

IV. PREPROCESSING ANALYSIS

In this section we will go deep into each of the preprocessing steps and understand how TensorFlow implements these steps. Following are the pre-processing steps used in the pipeline created for training a Resnet50 model:

A. Decoding JPEG Images

JPEG is an image compression algorithm, which is a lossy algorithm. The raw JPEG image is read from the disk, it doesn't directly provide the data in a format that is required by the machine learning models, which is actually an RGB format. So, a pre-processing step is required to process the input raw image to extract the RGB data from the image, which is a JPEG decoding process. The JPEG encoding/decoding process itself has lots of steps as illustrated in Fig 7.

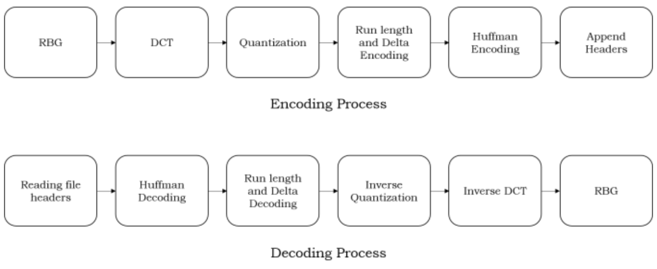


Fig. 7. JPEG Encoding/Decoding process.

Here in this section, we will emphasize only on encoding process of JPEG image, from which decoding process follows the exact reverse path in opposite direction:

a) Reading File Headers: Each JPEG file has several file markers that identify the start and end of various file sections, such as start/end of image file, start of quantization tables, start of Huffman tables etc. These markers are required to identify the start of each specific section of image, which is further used for recreating the image in RGB format. Each marker (16 bits), is immediately followed by the size of the section, which is also 16 bits in size. The following steps in the decoding process will make use of these sections from the input JPEG file.

b) Discrete Cosine Transformation (DCT): The whole encoding process is based on the fact human eyes cannot see the high frequency components in the images. So, the encoding process tries to compress the images by filtering the high frequency components from the image, and the image still looks similar to the naked human eyes. For analysis of the high frequency components in the input image, Discrete Cosine Transformation is applied to this image. The intuition behind using the DCT is that it converts each discrete pixel in the image to the combination of cosine waves that are used to perform the frequency analysis of the image. For this step, JPEG divides the input image into 8x8 chunks of pixels also known as Minimum Coding Units (MCUs). The discrete cosine transformation, shown below, is applied to the block of 8x8 pixels.

$$T_{i,j} = \left\{ \begin{array}{ll} \frac{1}{\sqrt{N}} & \text{if } i = 0 \\ \sqrt{\frac{2}{N}} \cos\left[\frac{(2j+1)i\pi}{2N} \right] & \text{if } i > 0 \end{array} \right\}$$

Fig. 8. Equation for computation DCT coefficients

So, DCT helps us to identify how we can reproduce the image block 8x8 matrix of cosine waves, i.e., how each of 64 cosine functions contributes to each pixel in the images. Fig. 9 shows the 8x8 matrix of cosine function. Fig. 9 clearly shows that the elements on the top left corner contain low frequency components and elements of the matrix on bottom right corner contain highest frequency components. So, the main focus here is to identify these high frequency areas in an image and suppress them using quantization.

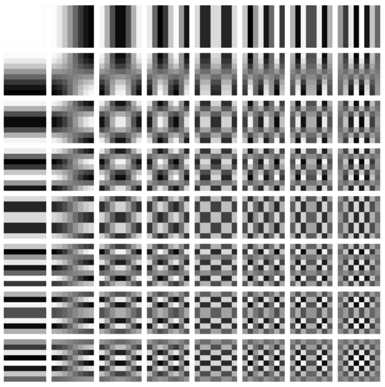


Fig. 9. 8x8 matrix for cosine functions.

After Identifying the high frequency components in the input image, these components can be eliminated by equating the high frequency pixel value equals to zero using quantization technique.

c) Quantization: After the DCT step, quantization is performed. Quantization, in image processing, is a compression technique that converts the range of values to a single quantum discrete value. Quantization is a way to suppress the high frequency components in the image. Quantization is done by an 8x8 quantization matrix (shown in Fig. 10). So, the level of quantization decides the amount of compression in the image. Most commonly used quantization filters are Q10, Q50 and Q90, which corresponds to 10%, 50% and 90% compression respectively. TensorFlow uses a Q50 quantization filter by default, but can be configured to use other filters as well.

$$\underset{\text{DCT Matrix}}{\begin{bmatrix} -415 & -33 & -58 & 35 & 58 & -51 & -15 & -12 \\ 5 & -34 & 49 & 18 & 27 & 1 & -5 & 3 \\ -46 & 14 & 80 & -35 & -50 & 19 & 7 & -18 \\ -53 & 21 & 34 & -20 & 2 & 34 & 36 & 12 \\ 9 & -2 & 9 & -5 & -32 & -15 & 45 & 37 \\ -8 & 15 & -16 & 7 & -8 & 11 & 4 & 7 \\ 19 & -28 & -2 & -26 & -2 & 7 & -44 & -21 \\ 18 & 25 & -12 & -44 & 35 & 48 & -37 & -3 \end{bmatrix}} \underset{\text{Quantization Matrix}}{\begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}}$$

Fig. 10. An example DCT Matrix and Quantization Matrix (Q50) used by TensorFlow.

Fig. 10 shows the quantization matrix used by TensorFlow. This matrix is used to do element wise division with the 8x8 DCT matrix obtained in the previous step of the process. The quantization filter has higher numerical values in the bottom right corner, which is the place in the DCT matrix with higher frequencies. So, the quantization process is suppressing the higher frequency elements and slows the lower frequency elements to stay. The quantized matrix obtained after the element wise division is shown in Fig. 11. After the element wise division of two matrices, the elements of the quantized matrix are converted to their nearest integer. Fig. 11 shows how the pixels with high frequency components are converted to zero.

$$\begin{bmatrix} -26 & -3 & -6 & 2 & 2 & -1 & 0 & 0 \\ 0 & -3 & 4 & 1 & 1 & 0 & 0 & 0 \\ -3 & 1 & 5 & -1 & -1 & 0 & 0 & 0 \\ -4 & 1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Quantized Matrix

Fig. 11. Quantized Matrix

d) Run-length and Huffman Encoding: After the quantization step, now the compressed image is ready for writing it in desired JPEG format. First of all, the 8x8 MCUs are converted into linear 1-D format, which is done in zig-zag fashion as shown in Fig. 12.

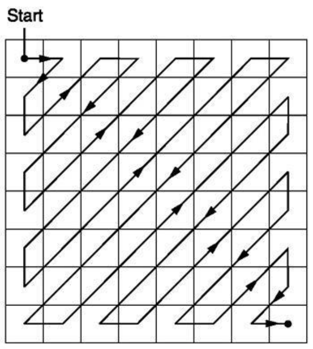


Fig. 12. Procedure to convert 8x8 matrix to 1-D matrix of size 64.

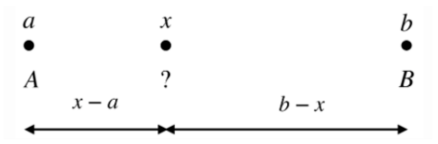
The main reason for using this zig-zag encoding is to get the low frequency components in the beginning of the 1-D array. This helps in further compression of the data using Run-length or Delta encoding. Run-Length encoding is the type of encoding process which is used for compression of repeated data. As we have made most of the high frequency components equal to zero, the 1-D matrix obtained after zig-zag encoding must have lots of zeros at the end. So, we can use Run-Length encoding in that place to save memory. Further compression is achieved by using Huffman Encoding technique.

The above process explains the encoding technique for JPEG images, the decoding process follows the exact opposite path. All the data required for decoding an image, such as Quantization filter, encoding method used, Huffman tables etc., is provided in JPEG format.

B. Image Resizing

Image resizing is done by using interpolation techniques to fill the new pixels in an image. There are various interpolation techniques used by the image processing community. Some common examples of interpolation techniques are Nearest Neighbor, Gaussian, Bilinear and Cubic interpolations. TensorFlow, by default, uses Bilinear Interpolation for resizing images to specific size. The main reason this method is used as a default method for resizing is that it is efficient (than nearest neighbor method) and computationally less expensive (than cubic method).

Bilinear interpolation is extension if linear interpolation in two dimensions. Fig. 13 shows linear interpolation technique. This technique helps to derive a value between two data points in such a way that the value gradually increases from one data point to the another one.



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

Fig. 13. Linear interpolation

The formula in Fig. 13 shows how linear interpolation is used to calculate the value of pixel at location *x* between two pixels *a* and *b*. So, when resizing images when extra pixels are added between the existing pixels, their values are calculated using this interpolation technique. When linear interpolation is applied in both x and y dimensions, its called bilinear interpolation and is used for resizing input images in two dimensional space.

Fig. 14 shows how the bilinear interpolation with a two-dimensional image array.

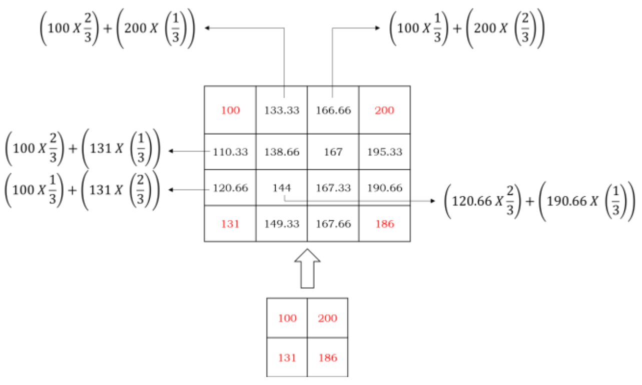

$(100\,X\,\frac{2}{3}) + (200\,X\,(\frac{1}{3}))$
$(100\,X\,\frac{1}{3}) + (200\,X\,(\frac{2}{3}))$
$(100\,X\,\frac{2}{3}) + (131\,X\,(\frac{1}{3}))$
$(100\,X\,\frac{1}{3}) + (131\,X\,(\frac{2}{3}))$
$(120.66\,X\,\frac{2}{3}) + (190.66\,X\,(\frac{1}{3}))$
100
133.33
166.66
200
110.33
138.66
167
195.33
120.66
144
167.33
190.66
131
149.33
167.66
186
100
200
131
186

Fig. 14. Bilinear Interpolation for resizing 2x2 array to 4x4 array.

Another technique, the nearest neighbor, is illustrated in Fig. 15. This technique is computationally efficient, as it fills the new added pixels with the value of the nearest neighbors to the pixels.

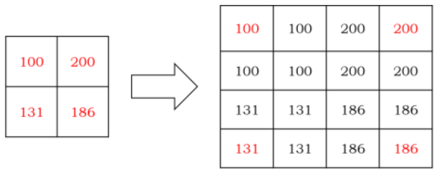


Fig. 15. Nearest Neighbor Interpolation

C. Random Brightness

After performing the above two steps in the preprocessing pipeline, the third step is to add random noise to input images. This step is essential for the deep learning models that are required to perform well for images with variety of light settings. TensorFlow uses a random number generator to generate random values between a certain interval provided by the user. This random matrix is added to the actual image matrix obtained in previous step. Fig. 16 shows how TensorFlow generates a random brightness matrix within a range [-32/255, 32 /255] and then adds it to image pixel matrix.

The red pixel values in Fig. 16 shows the values that overflowed due to addition of random noise. In the last pre-processing step, these values are clipped to make sure that all pixel values remain in interval [0, 1].

$$\underset{\text{Image Pixel Matrix}}{\begin{array}{|c|c|c|c|}\hline 0.392 & 0.379 & 0.102 & 0.996 \\ \hline 0.174 & 0.450 & 0.000 & 1.000 \\ \hline 0.803 & 1.000 & 0.447 & 0.984 \\ \hline 0.832 & 0.012 & 0.473 & 0.226 \\ \hline \end{array}} + \underset{\substack{\text{Random Brightness Matrix} \\ \text{(-32/255) to (32/255)}}}{\begin{array}{|c|c|c|c|}\hline 0.125 & -0.026 & -0.004 & -0.103 \\ \hline 0.012 & -0.114 & 0.043 & 0.105 \\ \hline -0.121 & 0.019 & 0.085 & 0.119 \\ \hline 0.111 & -0.113 & 0.014 & -0.076 \\ \hline \end{array}} = \underset{\text{Final Image Matrix}}{\begin{array}{|c|c|c|c|}\hline 0.517 & 0.353 & 0.098 & 0.893 \\ \hline 0.186 & 0.336 & 0.043 & 1.105 \\ \hline 0.682 & 1.019 & 0.532 & 1.103 \\ \hline 0.721 & -0.101 & 0.026 & 0.15 \\ \hline \end{array}}$$

Fig. 16. Adding random brightness to image matrix

D. *Image Clipping*

This pre-processing step is required to normalize the pixel values in the image matrix. In this step, the pixel values are clipped if the values of the pixel are either greater than one or less than zero. This step is crucial in order to eliminate the exploding gradients problem. Fig. 17 illustrates the clipping procedure employed by TensorFlow when the clipping interval provided is [0, 1]. Clipping interval is configurable by user and must be explicitly mentioned during API call else [0, 1] is assumed by default.

$$\underset{\text{Image Matrix}}{\begin{array}{|c|c|c|c|}\hline 0.517 & 0.353 & 0.098 & 0.893 \\ \hline 0.186 & 0.336 & 0.043 & 1.105 \\ \hline 0.682 & 1.019 & 0.532 & 1.103 \\ \hline 0.721 & -0.101 & 0.026 & 0.15 \\ \hline \end{array}} \Rightarrow \underset{\text{Clipped Image Matrix}}{\begin{array}{|c|c|c|c|}\hline 0.517 & 0.353 & 0.098 & 0.893 \\ \hline 0.186 & 0.336 & 0.043 & 1.000 \\ \hline 0.682 & 1.000 & 0.532 & 1.000 \\ \hline 0.721 & 0.000 & 0.026 & 0.15 \\ \hline \end{array}}$$

Fig. 17. Clipping pixel values in the interval [0, 1]

V. EVALUATION OF INPUT DATA PIPELINE

A. *Hardware and Software*

This project focuses on the analysis of how TensorFlow implements the input data pipelines and how we can offload the certain sections of this entire pipeline to FPGAs inside SmartSSD to efficient data models for deep learning. For this project, we are using an AWS instance to analyze the deep learning models. The AWS instance used is *f1.2xlarge*, that contains four Intel(R) Core(TM) i7 8564U CPU cores and contains one Nvidia Tesla V100-SXM2-16GB GPU. This GPU has 16GB global system memory, supports both L1 (128 KB per Streaming Multiprocessor) and L2 caches (6 MB), 5120 CUDA cores and supports CUDA 7.0 (with OpenCL 1.2). The AWS system has an SSD attached to it via SATA interface, that has storage capacity of 80 Gigabytes which is sufficient for our experiments.

From a software point of view, AWS instance has Linux operating system with Ubuntu 20.04 distribution installed on it. Apart from that, we have installed TensorFlow version 2.4.1, that has inbuilt capability to support Nvidia GPU initialization and kernel execution. To fulfill the dependency requirements for TensorFlow, we have also installed CUDA® 11.0 GPU toolkit, cuDNN 8.0.4, which is a Deep Neural Network library required for optimized execution of TensorFlow on GPUs. Tensorboard v2.4.1 is also installed for analysis of performance metrics that are generated by TensorFlow.

B. *Methodology*

In this project, the complete objective analysis is done using the simulation of the Resnet50 model in TensorFlow, that uses input data pipeline. The Resent50 model is trained using ImageNet dataset that has 1,281,167 training images, 50,000 validation images and 100,000 test images that are classified in over 1000 object classes.

For input data pipeline, the we are performing four major pre-processing steps, which are decoding JPEG image format, resizing the image to 200 x 200 x 3 dimension, adding random brightness to the image in interval of range [-32/255, 32/255] and clipping the pixel values to get values in range [0, 1]. Also, for this project we varied the total number of worker threads by varying the parameter *num_parallel_calls* from 1 to 16. Next subsections will discuss the complete analysis and observation captured during the project.

C. *Performance of Resnet50 model*

The Resnet50 model works with input data pipelines. We have tested this model with ImageNet dataset. TensorFlow uses the input pipeline to feed pre-processed data to the model that is computed on the Nvidia GPU. To check the accuracy achieved by the training set with the Resnet50 model, we have tested the model with different batch sizes and epochs to see the effect of batch size and number of epochs on the training accuracy. Graph in Fig. 18 shows the accuracy of the Resnet50 model with various batch sizes for thirty training epochs. We are only able to increase the batch size to 160 as this the batch size that GPU is able to support with the given dataset. Going beyond 160 images in one batch causes memory issues in GPU and causes GPU kernel to exit without properly executing.

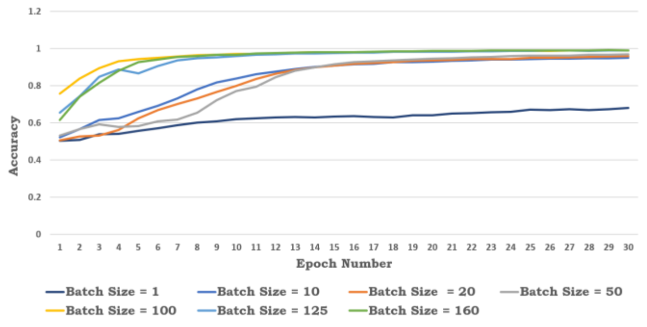


Fig. 18. Training accuracy achieved by Resnet50 Model with different batch sizes for 30 training epochs

Fig. 18 clearly illustrates the impact of batch size on the training accuracy of the model. The smaller batch sizes tend to overfit the model. This is because the model changes its weights after analyzing a smaller number of images, thus learning a lesser number of features in a fixed number of epochs. As we increase the batch size, the graph shows that training accuracy keeps on increasing. So, by increasing the batch sizes, we can achieve higher accuracy in a lesser number of epochs. This can be clearly seen in the graph, as batch sizes greater than 50 achieve accuracy of about 90% in around 12th epoch as compared to the smaller batch sizes that achieve the same amount of accuracy in 30 epochs.

D. *Performance of Input Data Pipeline*

The analysis of input data pipelines is crucial to understand the input pipeline and to narrow down the bottlenecks in the data pipeline. We created a scenario to measure the time spent by batches of different sizes in the input pipeline.

To measure the performance of the input data pipeline, we must make sure that TensorFlow fetches images directly from the SSD. To create this scenario, we made sure that no image data is present in the OS page cache, by flushing page cache using following command:

sync && echo 3 > /proc/sys/vm/drop_caches.

Graph in Fig. 19 shows the time taken by one batch of different sizes inside the pipeline, which essentially includes the time taken by TensorFlow to fetch the data from SSD and time taken to pre-process the whole batch. Graph also compares the time taken by different batches when TensorFlow employs more worker threads, in this case 4. As mentioned in section II subsection B, this is achieved in the data pipeline by setting *num_parallel_calls* argument to four.

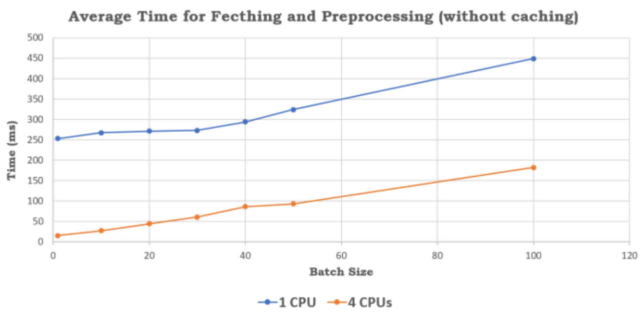


Fig. 19. Average time spent by a batch in input data pipeline

The above graph shows that the increase in the number of worker threads increases the efficiency of the input pipeline. This graph proves that SSD is not the only bottle neck in the system, however the CPU also acts as a bottleneck in the input pipeline.

To get a little more insight into the performance of each pre-processing stage in the input pipeline, we used TensorFlow's Input Pipeline Analyzer. This profiler provides efficient timing analysis of the time spent in each pre-processing stage of the pipeline. TensorFlow provides a callback API *tf.keras.callbacks.TensorBoard* to capture the profiling data during training runtime. This API is used to set various profiling settings and can be passed as a callback function to training API, and is called after each epoch to log the profiling data.

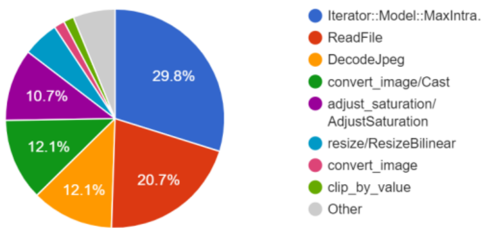


Fig. 20. Breakup of share of time taken by each pre-processing stage

The above graph shows that about 50% of CPU time is spent on the pre-processing section of the data pipeline. This is a huge amount of time that is spent for pre-processing of the data. There are various ways to reduce this time, say by preprocessing the data independently before the training. This can be done if the input data is pre-processed offline or use the dedicated hardware for pre-processing which can speed up the whole training process to a great extent.

E. *Compression of Dataset*

As the graph in Fig. 20 shows that around 20% of CPU time is consumed in reading the input raw image files from the storage media. As we are reading the raw image files that are on an average 50K in size each. However, the research has proved that compression of dataset can result in reducing the disk IO latency and can speed up the overall fetching process in the input data pipeline.

TensorFlow has an inbuilt file format known as TFRecords, which is a file format used for storing the sequence of binary strings. TensorFlow uses Protocol Buffer to implement the TFRecord functionality. TensorFlow provides users with the capability to create custom serialization formats. For this purpose, TensorFlow provides two APIs, *tf.train.Example* and *tf.train.SequenceExample*. These two format APIs are implemented as Protocol Buffers, and the data must be stored in one of these two structures. Serialized file, i.e. The TFRecord file is then written to the disk using API *tf.python_io.TFRecordWriter*. Protocol Buffers are a cross-platform, cross-language library for efficient serialization of structured data. Converting input data into TFRecords format has following advantages:

1. As serialization of data results in compression of data, TFRecords consume less space than original dataset. So, TFRecords are more memory efficient.

2. TensorFlow has capability of reading the TFRecords with parallel IO operations. Also, as the size of TFRecord is smaller than actual data, more data can be fetched in a single IO command.

Graph in Fig. 21 shows the improvement in average fetching and preprocessing times for each batch when TFRecords are used instead of raw images. For this purpose, each batch is converted into one TFRecord file, so each TFRecord file contains serialized information of various raw images.

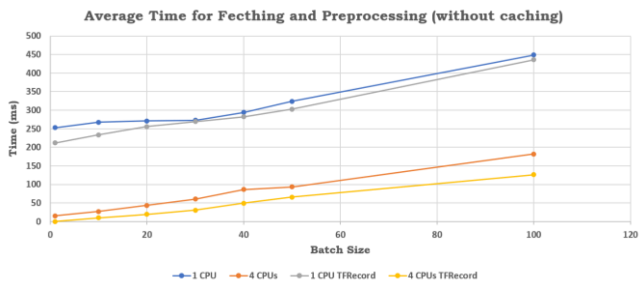


Fig. 21. Breakup of share of time taken by each pre-processing stage

F. *Performance with preprocessed input data*

For this project, our hypothesis is to increase the input pipeline performance by moving the preprocessing steps to dedicated FPGA in SmartSSD. To verify this hypothesis, we have independently preprocessed our input data, such that our pipeline no longer needs to perform any preprocessing steps except from fetching the preprocessed data only. This is done by performing the preprocessing steps, such as Decoding JPEG image, resizing image to dimensions 200 X 200 X 3 and adding random brightness, offline on raw JPEG input images.

Fig 22. shows the performance enhancement gained when the preprocessed data is used as input the data pipeline in place of raw images. This can be attributed to offloading of CPU intensive preprocessing operations (discussed in Section V subsection D) as the preprocessing steps are taking about 50% of the overall time spent in pipeline execution.

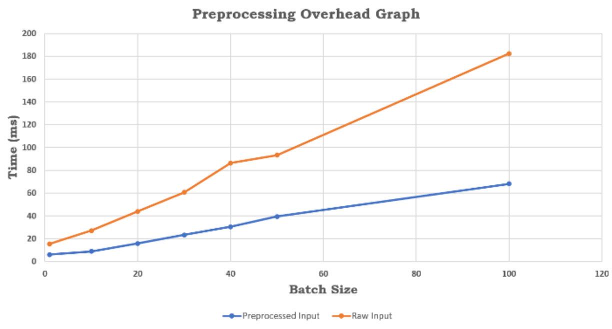


Fig. 22. Performance comparison between raw input data and preprocessed input data.

This graph definitely provides us confidence on the validity of our hypothesis that we can remove the CPU bottleneck by offloading the preprocessing steps to much faster dedicated hardware (FPGAs).

VI. OPTIMIZATIONS

This section will go into details of the ideas and various other optimizations that are thought of as the part of this analysis project. These new ideas such as how beneficial it is to offload the pre-processing section of the pipeline to FPGA inside SmartSSD, how to integrate such a system with TensorFlow and some other ways to reduce the pre-processing times in the pipelines.

A. Offloading Preprocessing steps to FPGA in SmartSSD

For understanding how we can offload the preprocessing steps to the FPGA, we have implemented code for three main preprocessing steps, which are JPEG decoding, resizing image and adding random brightness to image. The whole code is implemented in C language which can be easily integrated with the FPGA kernel APIs such as OpenCL. For the implementation of JPEG decoder, we have used the lightweight decoder implementation, Pico Decoder, presented by Intel. This implementation is very efficient in extraction of the three-color planes which are Red, Blue and Green by decoding the compressed JPEG format. Once extracted, these planes are individually resized using the Bilinear Interpolation technique mentioned in section IV subsection C. All the resized planes are then arranged in a specific format that is identified by TensorFlow. This format is known as Tensor, where the dimensions of this format is (Number of Planes) X (Width of the Image) X (Height of the Image). Essentially, TensorFlow requires all the pixel values (in an RGB plane) for each pixel to be together in one dimension of Tensor.

One thing that we have analyzed while creating a tensor is, if we convert the pixel values from integer to float (for normalization), the total size of the tensor increases by four times that will eventually create the network or I/O overhead. This is because of the reason that each pixel is represented using 8 bits, as it contains values from 0 to 255. However, for representing a normalized pixel value from 0 to 1, we need floating numbers that at least need 32 bits to be stored in memory.

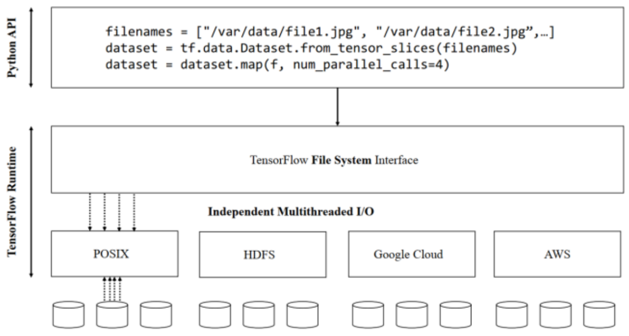
B. Custom File System

TensorFlow has a file system module [3], which is working behind all the I/O calls made through TensorFlow. As of now, TensorFlow file system module supports following file systems:

1. A standard POSIX filesystem
2. HDFS - Hadoop File System
3. GCS - Google Cloud Storage filesystem
4. S3 - Amazon Simple Storage Service filesystem

The block diagram for TensorFlow File System module is illustrated in Fig. 23.

TensorFlow has set up a URI (Uniform Resource Identifier) scheme, which helps TensorFlow to identify which type of file system is being requested by the user. Fig. 24 illustrates how TensorFlow identifies various file system queries by user.



```
filenames = ["/var/data/file1.jpg", "/var/data/file2.jpg",…]
dataset = tf.data.Dataset.from_tensor_slices(filenames)
dataset = dataset.map(f, num_parallel_calls=4)
```

Python API
TensorFlow Runtime
TensorFlow File System Interface
Independent Multithreaded I/O
POSIX
HDFS
Google Cloud
AWS

Fig. 23. TensorFlow File System Interface

TensorFlow file system can be divided into three main layers of APIs:

1. Low-Level API: These APIs directly handle all the IOs and hide all the complexities of read/write calls from the user.
2. Convenience API: These APIs are used to implement buffering operations and provide the additional functionality of on-the-fly serialization/ deserialization of data.
3. High-Level API: These are the actual APIs or ops called from Python wrappers provided by TensorFlow.

```
tf.io.gfile.GFile("/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("https://my.website/path/to/file")
# URI scheme:     |<--->|
# URI host:             |<----------->|
# URI path:                           |<---------->|
```

Fig. 24. URI Scheme used by TensorFlow

As, for using the SmartSSD, we need to create our own implementation of the filesystem for reading and writing data to and from the SmartSSD. So, we need to integrate this custom implementation of Filesystem with the TensorFlow to enable the model training using the data retrieved from SmartSSD. For this very purpose TensorFlow provides a feature for users to implement a filesystem separately and to load the custom file system during the runtime. This feature is very robust for creating custom file systems that are efficient from a specific application perspective, be it deep learning applications or data analytics applications. TensorFlow provides a robust set of test cases as well for testing the custom file system before loading it during the runtime. TensorFlow identifies the new custom filesystem implementation using the URI assigned by the user to the custom file system.

This type of custom filesystem is very appropriate for this project as we are trying to offload the whole pre-processing section of the input pipeline to FPGA inside the SSD. To make use of the FPGA subsystem inside the SSD, there is a requirement of a new filesystem that can speed up the I/Os by leveraging the FPGA subsystem. So, with TensorFlow providing this feature to load a custom file system during runtime will resolve the problem of re-compiling the whole TensorFlow code to a custom implementation.

C. *Offloading Preprocessing steps to GPU*

The graph in Fig 25 shows that about 50% of CPU time is spent on the pre-processing section of the data pipeline. One way of reducing the pre-processing time is to off-load pre-processing to the GPU. For this purpose, the Nvidia DALI Data Loading Library is used. This library is capable of data loading and pre-processing data using GPUs for acceleration of deep learning model training.

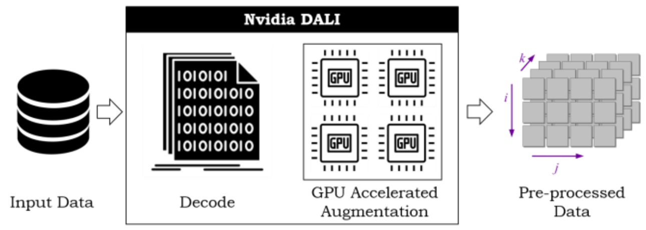


Fig. 25. Nvidia DALI Pipeline

It provides a collection of highly optimized building blocks for loading and processing image, video and audio data. It can be used as a portable drop-in replacement for built in data loaders and data iterators in popular deep learning frameworks. DALI addresses the problem of the CPU bottleneck by offloading data preprocessing to the GPU. Fig. 25 shows the Nvidia DALI execution path as it integrates inside the input pipeline Additionally, DALI relies on its own execution engine, built to maximize the throughput of the input pipeline. Features such as prefetching, parallel execution, and batch processing are handled transparently for the user. TensorFlow has a plugin to include the Nvidia DALI implementation into the input data pipeline.

VII. SUMMARY

The purpose of this project is to understand the machine learning frameworks from algorithmic point of view and to understand how these algorithms use the underlying hardware capabilities. We dived deep into an instance of input data pipeline that we created to train Resent50 model. During the analysis of this input pipeline, we found out that these input pipelines, although are expected to increase the overall efficiency of the system, but because of their I/O and CPU intensive nature they may create bottlenecks in the data loading path of ML model. These bottlenecks eventually lead to data starvation of resources such as GPU and increasing the model training time. In this project we also practically analyzed the scenario, where if the preprocessing is offloaded to FPGA in SmartSSD, how the training times can be improved by considerable amount. Not only the CPU bandwidth is saved by efficiently utilizing the FPGAs, however network bandwidth can be saved to a huge extent as preprocessed data is very small as compared to the original image. This analysis is very crucial to integrate the emerging storage technologies, such as SmartSSDs, with the machine learning models to enable the very efficient data intensive model training.

REFERENCES

[1] Mohan, Jayashree & Phanishayee, Amar & Raniwala, Ashish & Chidambaram, Vijay. (2021). Analyzing and mitigating data stalls in DNN training. Proceedings of the VLDB Endowment. 14. 771-784. 10.14778/3446095.3446100.

[2] He, Kaiming et al. “Deep Residual Learning for Image Recognition.” 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2016): 770-778.

[3] S. W. D. Chien et al., "Characterizing Deep-Learning I/O Workloads in TensorFlow," 2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS), Dallas, TX, USA, 2018, pp. 54-63, doi: 10.1109/PDSW-DISCS.2018.00011.

[4] Zolnouri, Mahdi & Li, Xinlin & Partovi Nia, Vahid. (2020). Importance of Data Loading Pipeline in Training Deep Neural Networks.

[5] Abadi, Martín, P. Barham, J. Chen, Z. Chen, Andy Davis, J. Dean, M. Devin, Sanjay Ghemawat, Geoffrey Irving, M. Isard, M. Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, D. Murray, Benoit Steiner, P. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Y. Yu and Xiaoqiang Zhang. “TensorFlow: A system for large-scale machine learning.” OSDI (2016).

[6] A. Aizman, G. Maltby and T. Breuel, "High Performance I/O For Large Scale Deep Learning," 2019 IEEE International Conference on Big Data (Big Data), 2019, pp. 5965-5967.