Demystifying the TensorFlow Eager Execution of Deep Learning Inference on a CPU-GPU Tandem

Paul Delestrac, Lionel Torres, David Novo

Demystifying the TensorFlow Eager Execution of Deep Learning Inference on a CPU-GPU Tandem

Paul Delestrac, Lionel Torres, and David Novo
LIRMM, University of Montpellier, CNRS
Montpellier, France
Email: {firstname}.{lastname}@lirmm.fr

***Abstract*—Machine Learning (ML) frameworks are tools that facilitate the development and deployment of ML models. These tools are major catalysts of the recent explosion in ML models and hardware accelerators thanks to their high programming abstraction. However, such an abstraction also obfuscates the run-time execution of the model and complicates the understanding and identification of performance bottlenecks. In this paper, we demystify how a modern ML framework manages code execution from a high-level programming language. We focus our work on the TensorFlow eager execution, which remains obscure to many users despite being the simplest mode of execution in TensorFlow. We describe in detail the process followed by the runtime to run code on a CPU-GPU tandem. We propose new metrics to analyze the framework's runtime performance overhead. We use our metrics to conduct in-depth analysis of the inference process of two Convolutional Neural Networks (CNNs) (LeNet-5 and ResNet-50) and a transformer (BERT) for different batch sizes. Our results show that GPU kernels execution need to be long enough to exploit thread parallelism, and effectively hide the runtime overhead of the ML framework.**

***Index Terms*—ML frameworks, TensorFlow eager execution, profiling, CPU-GPU tandem**

I. Introduction

Deep Learning (DL) has gained massive popularity in many Machine Learning (ML) application domains such as computer vision (e.g., image classification), or natural language processing (e.g., language models). As a result, recent years have seen an explosion of new DL models (e.g., Convolutional Neural Networks (CNNs) [1] [2], transformers [3], etc.) and accelerator architectures (e.g., GPUs, TPUs [4], neural processing engines [5] [6], etc.). Open-source ML frameworks, such as TensorFlow [7], PyTorch [8] or MXNet [9], are major catalysts of such an explosion. An ML framework is a tool that facilitates the development and deployment of ML models. It provides a high-level interface (typically in Python) to describe the ML model and orchestrates its execution. Modern ML frameworks automatically distribute the model execution across CPUs and available accelerators, such as GPUs.

The high abstraction of ML frameworks greatly enables application developers to focus on the functionality of ML models without worrying about low-level implementation details. But on the flip side, a big disparity in performance can be observed between the different frameworks [10]. Additionally, such an abstraction obfuscates the run-time execution of the model and complicates the understanding and identification of performance bottlenecks. For example, TensorFlow, one of the most popular ML frameworks, offers two different modes of execution: *eager* and *graph* execution. Eager execution [11] is an imperative interface that executes operations immediately as they are called from Python. This enables fast debugging with immediate run-time errors. Therefore, eager execution is useful in the development phase of ML models. However, it is restricted to operation-level optimizations, which limits its performance. Graph execution extends the optimization scope by transforming the Python code into a graph of operations, which it optimizes before execution [12]. Therefore, graph execution is useful in the deployment phase as it achieves higher performance at the expense of debuggability. TensorFlow also supports different hardware backends, including CPU, GPU, TPU, etc. Due to this complexity, the TensorFlow codebase has grown to more than 3 million lines of code, which is practically inaccessible to most users despite being open source [13]. As a result, users often operate TensorFlow as a black box and cannot harness the full power of the framework. We believe this to be a common trend in the community with regard to modern ML frameworks.

Our **main goal** is to demystify how a modern ML framework manages code execution from a high-level programming language. To this end, we focus on the TensorFlow eager execution, which remains somewhat of a mystery to many users despite being the simplest mode of execution in TensorFlow. We analyze and describe how TensorFlow transforms high-level Python code into execution kernel calls to the GPU and CPU, when memory is allocated and what triggers data transfers between CPU and GPU. Furthermore, we leverage the insights gathered during the analysis to profile the overhead of the framework in the execution of representative ML inference models. To this end, we develop *TensorFlow eager runtime profiler*, a tool that extends the profiling tools provided by TensorFlow. We open source this tool in our GitLab repository [14].

In summary, this work makes the following contributions:

- We provide a detailed description of the main steps followed by the TensorFlow eager execution runtime to run high-level Python code on a CPU-GPU tandem.
- We propose new metrics, such as the scheduling queue occupation, to expose and analyze the ML framework's runtime performance overhead.
- We conduct in-depth profiling of the inference process of two CNNs (LeNet-5 [1] and ResNet-50 [2]) and a trans-

former (BERT [3]) for different batch sizes. Our results show that GPU kernels execution need to be long enough to exploit thread parallelism, and effectively hide the runtime overhead of the ML framework.

II. Background on GPU execution

NVIDIA GPU architecture [15] is designed to execute thousands of threads at the same time, sacrificing single-thread performance and execution latency to achieve high compute throughput on data-parallel workloads. A GPU needs a CPU managing the computing tasks. Together they form a *host/device* pair (CPU-GPU tandem). To execute code on GPU devices, NVIDIA exposes a programming platform and model called CUDA [16]. CUDA provides a programming language that extends C++ to program several software abstractions that allow the concurrent execution of GPU functions called *kernels* in multiple *streams*.

A *kernel* is a function callable from the CPU that runs on the GPU. The programmer specifies the execution configuration of a kernel in the source code. This configuration specifies the number of threads running concurrently on the GPU as well as the memory allocation and data transfer operations (i.e., `MemCpyH2D` to transfer from host to device and `MemcpyD2H` for the reverse operation) needed to run the kernel. It is also possible to specify the stream on which to run the kernel.

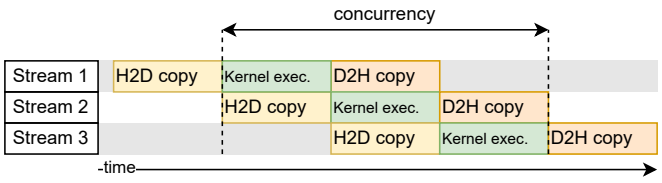


Fig. 1: CUDA Streams

Streams are a CUDA programming model feature where different work can be submitted to multiple queues and processed independently by the GPU. As illustrated in Fig. 1, the use of streams enables concurrency between data transfer and kernel execution. CUDA defines a *stream* as a a sequence of operations executing sequentially in the order as issued from the host CPU. However, operations issued in separate streams can be executed concurrently and can overlap. In concrete, CUDA exposes the following operations as independent tasks that can operate concurrently with one another:

- Computation on the host
- Computation on the device (i.e., kernel execution)
- Memory transfers from the host to the device (H2D copy)
- Memory transfers from the device to the host (D2H copy)
- Memory transfers within the memory of a given device
- Memory transfers among devices

TensorFlow has a very specific design for using CUDA streams. There is a main *compute* stream, a main pair of *host_to_device* and *device_to_host* streams, as well as a vector of *device_to_device* streams (used when computations are distributed over multiple devices). The main compute stream can handle transfers and computations. Secondary compute streams can be used, however, they cannot perform computations concurrently with the main stream.

III. TensorFlow eager execution

TensorFlow runtime is responsible for executing a user-defined model on a selected device, such as a CPU or a GPU. For the user to be able to build ML models, TensorFlow provides several APIs (e.g., JavaScript, C++, Java, etc.). The most complete, best documented, and most popular of them is the TensorFlow Python API. As discussed previously, the TensorFlow runtime can follow two execution modes: *eager execution* and *graph execution*. We will focus on TensorFlow eager execution runtime.

TensorFlow eager execution is the default mode of execution of TensorFlow. It executes operations immediately as they are called from the user-defined model. This behavior enables line-by-line debugging, which is not as straightforward when operations are buried in an optimized graph (i.e., when using graph execution). TensorFlow eager execution follows the same specific steps to execute each operation. We group the steps in three different phases: enqueuing, dequeuing and kernel execution (illustrated as ①, ②, and ③ in Fig. 2, respectively). TensorFlow eager execution can execute *synchronously* (SYNC) or *asynchronously* (ASYNC). In the case of SYNC execution, the enqueuing and dequeuing phases are merged into a single thread which is synchronized with the device kernel execution. This single main thread starts with the call of an operation in Python, launches the execution on the device and waits for the return value before executing the next operation. In the case of ASYNC execution, the enqueuing and dequeuing phases are performed by separate threads. The enqueuing phase starts with the call of an operation in Python and pushes one or several computation *nodes* in the scheduling queue. Asynchronously, the dequeuing phase pops the nodes from the queue and manages their execution. A computation node is a software object gathering all the information needed to call kernel execution on the target device.

In this section, we provide a detailed description of the main steps that the eager execution runtime follows to execute an ML model on a CPU-GPU tandem.

A. TensorFlow operations

To compose a TensorFlow model in Python, the user uses the available TensorFlow operations (e.g., matmul, relu, etc.) as defined in the Python API [17]. TensorFlow operations are linked to optimized kernels: CPU kernels for every operation, and GPU kernels for GPU-compatible operations. These kernels are accessible from the Python API via wrapper functions, which are automatically generated for every registered operation when building TensorFlow from source.

TensorFlow includes a very complete set of already registered operations, but still allows developers to add custom operations [18]. To register a new operation in TensorFlow, developers must (1) define its interface by specifying its inputs, outputs, and attributes types and shapes in a C++ module, (2)

implement a CPU *OpKernel* for the operation, (3) implement a CUDA kernel if GPU compatibility is intended, and (4) link the previously defined kernel(s) with the specific operation.

B. Enqueuing: Python host program

When starting to execute the Python wrapper function of the operation, the runtime has to know which mode of execution has to be used to run. This user-configurable parameter is stored into the *execution context*. The context is a collection of configuration parameters used by the runtime to execute the Python wrapper functions, such as execution mode (i.e., eager mode or graph mode), device placement policy, policy to use when running an operation on a device with inputs which are not on that device, etc. The context is initialized at the start of the Python program execution and is referred to whenever the runtime needs these configuration parameters to make choices during the execution. Most of the data stored in the context is thread-local (which is important for the runtime to be thread-agnostic), so that multiple TensorFlow programs can safely run in parallel.

Fig. 2 shows the main steps in the enqueuing thread. The eager execution of an operation starts with the *canonicalization* ⓐ of its inputs to the *Tensor* data type. At this point, memory is allocated on the host to store the inputs after the conversion to tensors. A Tensor is a multi-dimensional array defined by three parameters: a data type, a shape, and the actual values. However, the TensorFlow runtime does not directly manipulate the Tensor data type, but interacts with it through a *TensorHandle* ⓑ. A TensorHandle represents a tensor (or a not-yet-computed *FutureTensor*) which lives (or will live) on a device. A TensorHandle is able to provide the shape and type of the tensor even if the actual value is yet to be computed. This mechanism allows the runtime to race ahead and continue parsing Python operations without having to wait for the results of the computations. As a result, the main Python thread can run in parallel with an execution thread (i.e., ASYNC execution) instead of stalling while the operation is being executed (i.e., SYNC execution). If the executed kernels are sufficiently heavy to run, the Python thread can race ahead of the device execution and keep the device utilization high by hiding its own latency. However, when the runtime needs a tensor value to take a decision (e.g., a data-dependent if-then-else statement), it will stall until the required value becomes available (see Section III-D for an example).

After securing the inputs, the runtime gathers all the remaining information needed for the operation to execute in an *Op* object ⓒ. The Op object includes the name of the operation to run (as a string), pointers to the input data (i.e., encapsulated in a TensorHandle), and operation-dependent attributes (e.g., if one of the inputs needs to be transposed before the operation). Then, the runtime selects a device to execute the operation according to the execution context (if a device is specified by the user) or following an internal heuristic to find the fastest available device. Based on the name of the operation, the type and shape of its inputs, and the target device, the runtime also selects the kernel (i.e., OpKernel) to execute ⓓ.

At this point, the runtime validates the placement of the inputs of the operation ⓔ. If the inputs are not placed in the device targeted to execute the operation, the runtime schedules the necessary data-transfer operations. In the case of ASYNC execution, the runtime creates a specific node to handle data transfer that can be stored in a scheduling queue. In the case of a *host to device data transfer* action, which is automatically inserted by the runtime, the input data corresponds to the input data of the original operation (i.e., TensorHandle), the target device is the same as in the operation, and the prompted OpKernel links to a CUDA *MemcpyH2D* call.

Once the input placement is verified and scheduled, the runtime schedules the operation itself ⓕ. First, it creates a TensorHandle for the output of the operation, which is a FutureTensor. This allows the runtime to schedule future operations taking this output as an input without having to wait for the actual Tensor value. Then, the runtime creates a second node object gathering the Op object, a pointer to the selected device, and the selected OpKernel.

In the case of SYNC execution, the runtime runs a node immediately and stalls while waiting for a return value. In the case of ASYNC execution, the runtime will insert the node in a scheduling queue and directly return control of the thread. This architecture enables a layer of parallelism between the enqueuing and dequeuing processes, which TensorFlow uses to try to hide the latency of the Python thread behind the executor thread.

C. Dequeuing: Executor thread

The nodes in the scheduling queue are dequeued by a separate *executor thread* following the enqueuing order. The executor thread stalls or executes a queued node depending on whether the targeted device is busy or not.

The execution of a node consists of two main steps: the memory allocation of the operation outputs ⓖ, and the call for kernel execution on the targeted device ⓗ. A kernel describes the computations to perform and can also allocate memory for intermediate results when needed. TensorFlow includes a rich set of optimized kernels implemented using external libraries such as Eigen [19] and cuDNN [20]. Eigen is a C++ template library for linear algebra that can generate kernels for multiple input data types and target devices using the same codebase. This library is particularly useful for complicated linear algebra operations, as it eliminates the need to write optimized code for every supported data type, shape, and device. cuDNN is a library developed by NVIDIA that provides GPU-accelerated primitives for deep learning such as convolution, pooling, normalization, activation layers, and tensor transformation.

D. Illustration example

In this section, we provide a simple example (see Listing 1) to illustrate the most important concepts described previously in the section. We consider the example to run in asynchronous (ASYNC) mode on a system with a local CPU host and a local CUDA-compatible GPU device.

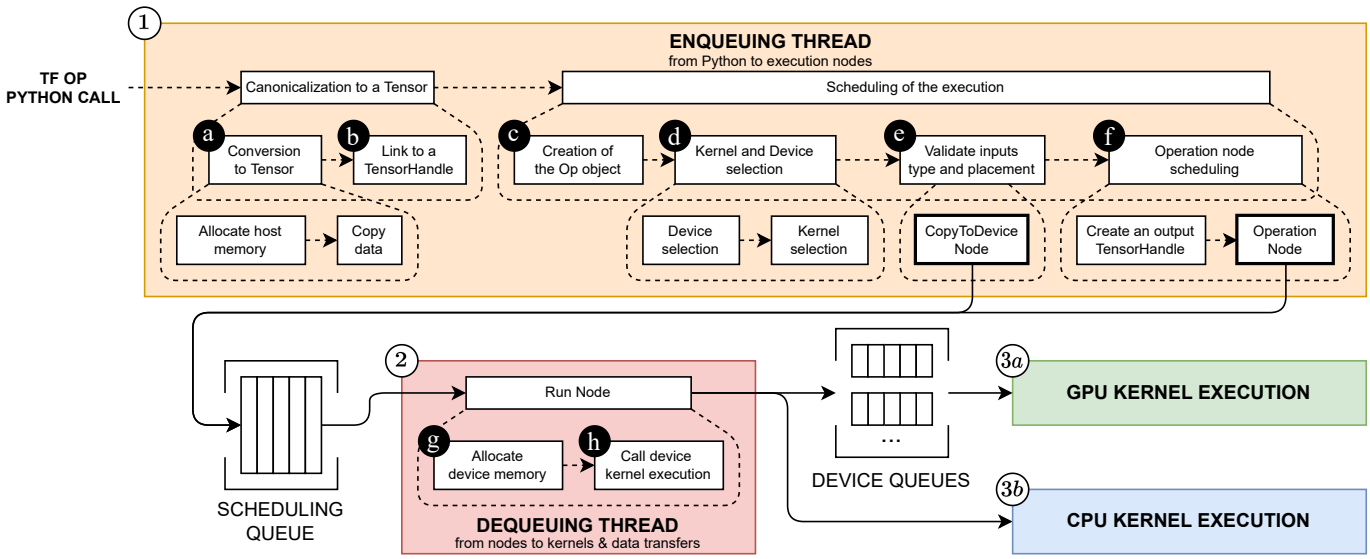


Fig. 2: Block diagram of the TensorFlow eager execution of a GPU-compatible operation in the ASYNC mode

```
1  import tensorflow as tf
2  import numpy as np
3
4  with tf.device("/GPU:0"):
5    x = np.random.randn(64, 64)
6    y = tf.constant(np.random.randn(64, 64))
7
8    z = tf.matmul(x, y, transpose_b=True)
9
10   output = tf.nn.relu(z)
11
12 print(output)
```

Listing 1: Illustration example including a sequence of operations using the TensorFlow Python API

The gist of our illustration example is a sequence of Python and TensorFlow operations which: creates a random matrix of size 64×64, creates a second random matrix of the same size which is explicitly converted to a constant Tensor, performs matrix multiplication on the two matrices, performs a ReLU activation on the matrix multiplication's output, and prints the result.

We now describe in detail how TensorFlow executes the code. Line 1 imports the TensorFlow package into the Python program. This import has the side effect of initializing the *execution context* which stores the configuration information needed by the runtime. By default, the execution mode specified in the context is eager execution. Line 2 imports the package *NumPy*, which is a popular Python package that offers mathematical functions to work with multidimensional arrays and matrices. In the example, it is used to create random inputs to feed the operations. Line 4 defines a TensorFlow scope (delimited by the `with` statement) that we use to specify the device on which we want to execute the operations. Then, this information is stored in the execution context and is referred to during the subsequent steps of the process. Line 5 uses the NumPy package to initialize a Python variable, storing a random matrix of shape 64×64 and of a data type defaulting to float64. This operation is independent of TensorFlow and does not invoke any TensorFlow runtime functionality.

Line 6 executes in multiple steps. The first step executes the same operation as Line 5: it creates a random matrix of shape 64×64 and of type float64. The second step is a call to the TensorFlow operation *tf.constant*. The execution of this operation is then taken up by the TensorFlow eager execution runtime. The runtime starts with the canonicalization of the input to a Tensor (a): it allocates host memory to store the resulting Tensor and copies the actual data of the matrix to the Tensor. The type and shape of the Tensor data follows the shape and type of the NumPy matrix: 64×64 and float64. Then, the Tensor is linked to a TensorHandle (b) that lives in the CPU. The runtime creates an Op object (c) which gathers: the operation name (i.e., *EagerConst* is the tf.constant operation name), the newly created TensorHandle, and no attributes (the operation does not have any). Next, the runtime selects the target device (GPU as specified in Line 4) and the kernel to run (*IdentityOp* for executing tf.constant in the GPU) (d). The runtime validates the placement of the input (e), which should also be in the GPU. However, the input TensorHandle currently lives in the CPU, so the runtime creates a *CopyToDevice* node to execute the required data transfer, and inserts it in the scheduling queue. Then, the runtime moves to the node scheduling of the tf.constant operation (f): creating a TensorHandle for the output (which lives on the GPU device) and the corresponding node object. Finally, the runtime inserts the node in the queue before returning the control to Python.

Line 8 follows a very similar process as Line 6 but with four key differences. First, the *tf.matmul* operation takes `x` and `y` as inputs. While `y` is already a Tensor, `x` is still a NumPy array. Hence, when converting the inputs to Tensors

ⓐ, the runtime prompts a tf.constant operation scheduling to ensure the conversion of `x` and its placement on the GPU. This tf.constant operation also triggers the creation of a TensorHandle ⓑ which lives on the GPU device. Second, the tf.matmul Op object ⓒ includes the `transpose_b` attribute, which is passed as an argument to the device kernel when executing the node. Third, the runtime selects the GPU as the execution device and chooses an OpKernel ⓓ that is linked to a CUDA kernel named `volta_dgemm_128x64_tn`. Fourth, no CopyToDevice node is needed as the two TensorHandles linked to the inputs already live in the GPU device.

Line `10` launches the execution of the *tf.nn.relu* operation, which follows the same process as the previous operation but with some simplifications. As its input is already a Tensor and lives on the GPU device, the runtime skips the conversion to Tensor ⓐ and the CopyToDevice node when validating the placement of the input ⓔ. This operation is linked to a GPU kernel named `Relu_GPU_DT_DOUBLE_DT_DOUBLE_kernel`.

Line `12` calls the Python `print` function on the Tensor `z`. This function is overloaded by TensorFlow: when the `print` function is called on a Tensor, the runtime uses the corresponding TensorHandle to know where the data is located. In this case, the data is a FutureTensor in the GPU. Accordingly, the runtime needs to copy the value of the Tensor `z` from the GPU to the CPU and convert it to a Python printable data type (i.e., NumPy array), which is then printed by Python. However, to execute this operation TensorFlow needs the actual value of the Tensor, which is only produced after the execution of the tf.nn.relu operation. As a result, the enqueuing thread stalls and waits for the value of the Tensor `z` to be produced.

In parallel with the enqueuing thread, the executor thread (i.e., dequeuing thread) has been processing the nodes in the scheduling queue. For each node, the executor thread allocates GPU memory for the output ⓖ and calls the corresponding GPU kernels ⓕ. Once all nodes have been dequeued and executed on the GPU device, the numerical value of the Tensor `z` is stored in the GPU memory. Thus, the enqueuing thread is notified that the value of `z` is available and resumes processing Line `12`. It sends a Memcpy command to copy the data from the GPU to the host, storing it as a printable data type (i.e., NumPy array). Finally, the Python interpreter prints the value of `z` in the user's console.

IV. Analysis

In this section, we present our approach to estimating the runtime performance overhead of TensorFlow eager execution using a target CPU-GPU tandem. We leverage TensorFlow profiling capabilities to analyze TensorFlow eager execution on a particular workload. From our analysis, we extract three key metrics: the share of the execution time spent in CPU or GPU kernel execution, the distribution of the execution time across the different phases of eager execution for each operation, and the scheduling queue utilization over time.

A. TensorFlow profiling capabilities

TensorFlow offers a profiling tool, the TensorFlow Profiler [21], to analyze the performance of an executed model. This profiler collects performance data to help understanding hardware resource utilization and identifying performance bottlenecks in the model. It collects profiling data sent by the TensorFlow runtime (e.g., the execution time of operations and functions of the runtime) and device metrics (e.g., kernel execution times, memory usage) recovered using the NVIDIA CUDA Profiling Tools Interface (CUPTI) [22]. Based on the collected data, TensorFlow offers different visualization tools (i.e., TensorBoard [23]) to gain insight into the input pipeline of the executed model, the distribution of TensorFlow operations between host and device, statistics about GPU and CPU kernels, memory profiling, and training statistics.

These tools provided by the profiler help to debug the model performance for inference and training. However, some of the information provided by these tools is often too low-level and difficult to interpret by the average user. As a result, it does not offer an accessible analysis of some of the inner mechanisms of the framework. Thus, in this section, we propose to extend the profiler with new analysis metrics to gain further understanding on how the framework optimizes and schedules the execution of operations. Fortunately, the data gathered by the profiler are saved in local JSON files during execution. Therefore, we can repurpose these data to provide new metrics and analysis to highlight how the framework optimizes and schedules the execution of operations. We open source these extensions [14] and hope to help average users harnessing the full power of the TensorFlow eager runtime.

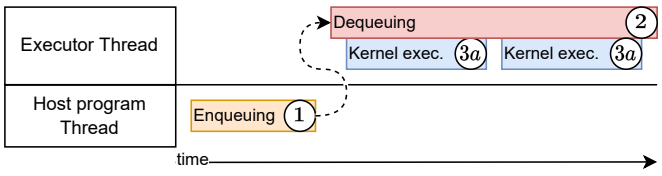
B. Time spent in kernel execution

For a TensorFlow program to achieve high performance, the time spent executing CPU and GPU kernels, which correspond to the real operations in the ML model, should dominate the total execution time. In practice, however, an ML runtime also needs to execute other enabling tasks such as parsing the Python code, copying data between CPU and GPU, etc.

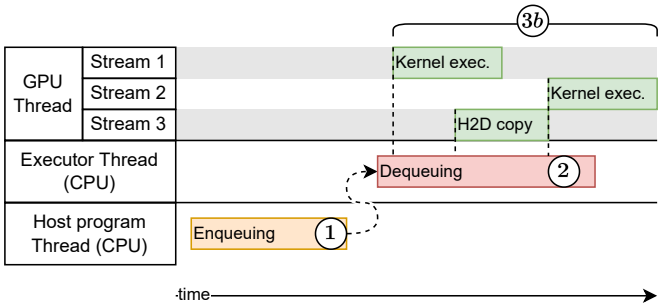
Our first analysis goal is to provide insight into how execution time is divided between kernel execution and the rest, which we consider to be the overhead of the runtime. To this end, we evaluate the distribution of execution time by parsing the data gathered by the TensorFlow profiling tool. We distribute the profiled events between three categories: CPU kernel execution, GPU kernel execution, and the rest of the execution. Additionally, when executing on a CPU-GPU tandem, the CPU and GPU kernel execution times can overlap, which is the best case scenario regarding computing resource utilization. Thus, we also track how often this happens.

C. Time distribution across eager execution phases

The TensorFlow eager execution of an operation (described in Section III) can be divided into three major phases as shown in Fig. 2: *enqueuing*, *dequeuing*, and *kernel execution*. Fig. 3 illustrates the phases for two different scenarios: (a) CPU-only execution, and (b) CPU-GPU tandem execution.



(a) CPU only execution



(b) CPU-GPU tandem execution

Fig. 3: Phases of TensorFlow eager execution (ASYNC)

First, the TensorFlow runtime handles the groundwork of execution: collecting the input data and attributes of the operation, selecting a device on which to execute and a corresponding kernel, and packaging everything into a node. This node will be either executed directly by the same enqueuing thread (SYNC mode) or queued for later execution by a distinct executor thread (ASYNC mode). These two threads are both threads running on host. We define this first part of the execution as the *enqueuing time* (① in Fig. 3).

Second, the executor thread runs once there is a node in the scheduling queue and handles the last steps before the kernel execution: allocating memory for the output, instantiating the kernel to execute, and calling the kernel execution on the targeted device. We define this second phase of the operation execution as the *dequeuing time* (② in Fig. 3). In the case of CPU kernel execution, both dequeuing and CPU kernel execution are performed by the executor thread. However, we separate CPU kernel execution time (③a in Fig. 3a) from the rest of the dequeuing (② in Fig. 3a).

Finally, the executor thread issues the corresponding kernel(s) to the targeted device for execution (③a and ③b in Fig. 3a and 3b, respectively). As our host/device execution target is a CPU-GPU tandem, this third phase of the execution corresponds to GPU kernel execution, for most of the operations. However, some operations have only a CPU kernel (see Section III-A). In such case, the execution corresponds to *CPU kernel execution time*. Thanks to the stream mechanism (described in Section II), data transfers between CPU and GPU can occur while a GPU kernel is executed.

The time spent enqueuing and dequeuing has to be shorter than the kernel execution to take advantage of the latency hiding mechanisms described in Section III. Thus, we also analyze the distribution of the execution times of each phase.

D. Utilization of the scheduling queue

TensorFlow eager execution includes a mechanism to hide node execution scheduling time: the *scheduling queue*. The last part of our analysis focuses on the scheduling queue utilization. To this end, we study the dequeuing phase and the kernel execution phase with two different metrics: scheduling queue utilization over time, and the distributions of the execution time between phases, when the scheduling queue is empty or not-empty (i.e., loaded with a node or more).

To profile the utilization of the scheduling queue over time, we use the rest of the profiling events (not sorted as kernel execution) at our disposal and distribute them in two new categories: enqueuing events and dequeuing events. We use these two categories to deduce the utilization of the scheduling queue over time. We assume that a node is in the queue from the end of the enqueuing phase until the beginning of the dequeuing phase. With this assumption, we are able to recreate the scheduling queue utilization over time. The role of the scheduling queue is to hide the node execution scheduling time. Thus, we cross-analyze this queue utilization with kernel execution timings and dequeuing timings. We consider three scenarios:

1) When CPU or GPU kernels are executing, the scheduling time is masked regardless of the utilization of the scheduling queue.
2) When the executor thread is active (i.e., is dequeuing a node from the scheduling queue), the scheduling time is masked regardless of the scheduling queue utilization.
3) When the executor thread is stalling, waiting for the enqueuing thread to insert a node in the queue, the scheduling queue is underutilized.

In the case of an empty scheduling queue, the main reason for the executor thread to be stalling would be that the host program (enqueuing thread) is waiting for some data to be computed by a kernel execution before scheduling a new node. For example, the dimensions of an operation $n[t]$ could depend on the result value of an operation $n[t-1]$, forcing the host program to wait for the value to be computed to be able to schedule the operation $n[t]$. To verify this hypothesis, we add the time when the enqueuing thread is stalling (i.e., waiting for a value) to our profiling of the scheduling queue utilization over time. In the case of a loaded scheduling queue (i.e., with one or more node), the main reason for the executor thread to be stalling would be that a data transfer is occuring. Hence, the executor thread is stalling for data to issue the next kernel call. To verify this hypothesis, we add the data transfers timings to our profiling of the scheduling queue utilization over time.

V. Results

In this section, we follow the approach described in Section IV to analyze TensorFlow eager execution on three different inference workloads: two Convolutional Neural Networks (CNN) (i.e., LeNet-5 and ResNet-50) and one transformer network (i.e., BERT).

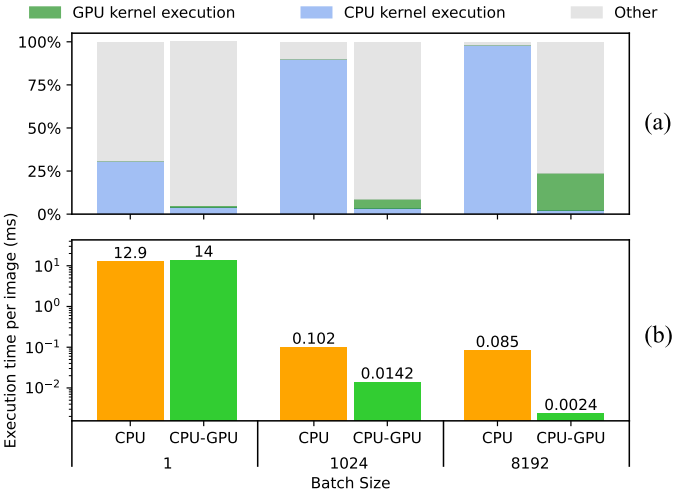


Fig. 4: LeNet-5 inference using three batch sizes on CPU-only and CPU-GPU tandem targets: (a) Execution time distribution; (b) Execution time per image

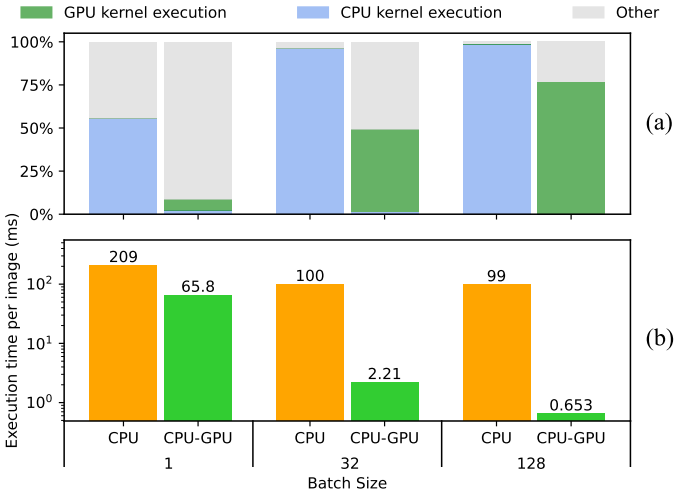


Fig. 5: ResNet-50 inference using three batch sizes on CPU-only and CPU-GPU tandem targets: (a) Execution time distribution; (b) Execution time per image

A. *Experimental setup*

To perform our analysis of the TensorFlow eager execution runtime, we performed inference with our three models (i.e., LeNet-5, ResNet-50 and BERT) on a CPU-GPU tandem (i.e., two Intel Cascade Lake 6248 with a total of 40 cores and 192GB of RAM, paired with an NVIDIA V100 with 32GB of dedicated RAM). We use TensorFlow [7] v2.8.0 as the target framework for our experiments.

The LeNet-5 model is recreated using TensorFlow, following the 1989 paper from LeCun et al. [1]. We use the ResNet-50 pre-trained model from the TensorFlow API Keras Applications [24], which follows the architecture described by He et al. [2]. The BERT pre-trained model is retrieved from the HuggingFace library [25] and follows the architecture described by Devlin et al. [3]. We execute the inference in the pre-trained models on TensorFlow with three representative batch sizes (one, medium, and high) chosen experimentally for each model. For each experiment, we run the inference once with the same batch size before profiling the execution. This enables us to compare our workloads with already cached OpKernels, avoiding irregularities (e.g., `red_zone_checker` kernel checks) in the execution time due to kernel instantiations.

B. *Kernel execution time*

We evaluate the share of the execution time dedicated to CPU and GPU kernel execution running on both our CPU-only target and our CPU-GPU tandem. We also evaluate the execution time per item (i.e., execution time divided by batch size) using the same workloads on the same platforms.

Fig. 4a, 5a, and 6a show the execution time distribution (between CPU kernel execution, GPU kernel execution and the rest) for LeNet-5, ResNet-50, and BERT, respectively. We make three observations. First, all the workloads show an increase in kernel execution share when increasing the batch

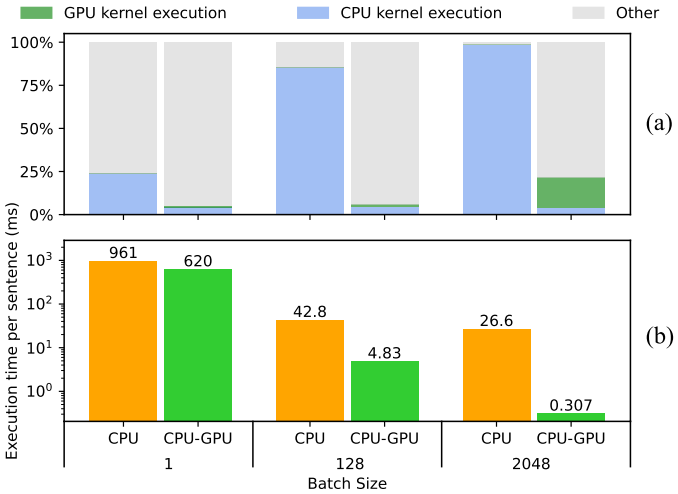


Fig. 6: BERT inference using three batch sizes on CPU-only and CPU-GPU tandem targets: (a) Execution time distribution, and (b) Execution time per sentence

size, both when executing on CPU-only and on CPU-GPU tandem. For example, on a CPU-GPU tandem, the LeNet-5 kernel execution time increases from 5% to 24% for a batch size of 1 and 8192 images, respectively; the ResNet-50 kernel execution time increases from 7% to 75% for a batch size of 1 and 128 images, respectively; and the BERT kernel execution time increases from 5% to 22% for a batch size of 1 and 2048 sentences, respectively. Second, the execution time of the CPU kernels becomes negligible when running on a CPU-GPU tandem. We observe that 3% of the execution time is spent in CPU kernel execution in average. In addition, the concurrent execution time between CPU and GPU kernel executions, which we include as part of the GPU kernel execution time in the figure, represents less than 1% of the total execution time. This shows that TensorFlow manages to use the GPU for most

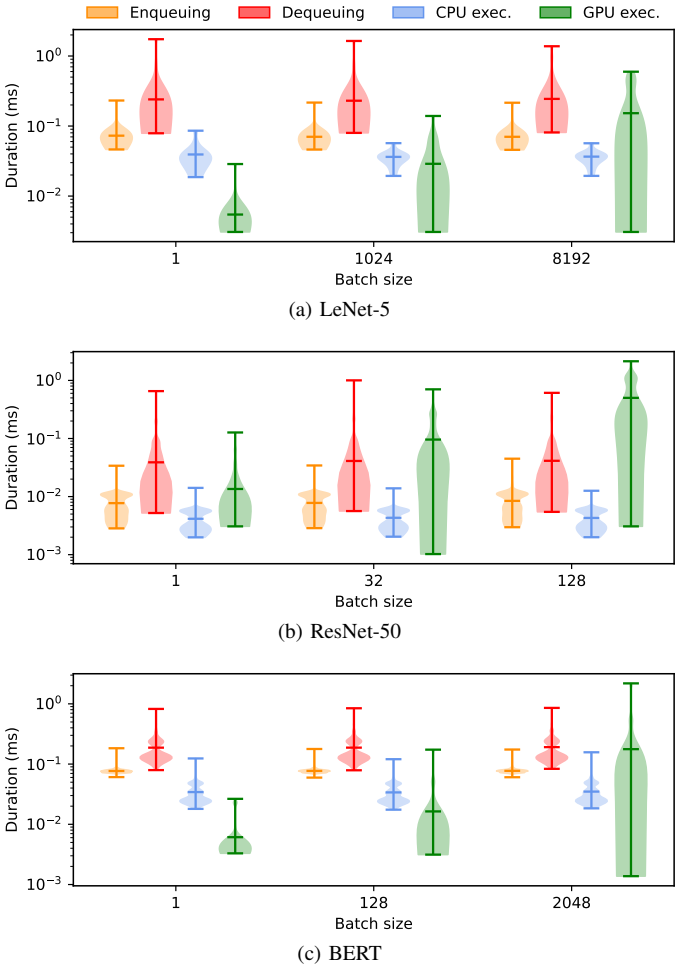
of the kernel execution when computing on large batch sizes. However, for small batch sizes, we can see that CPU kernel execution time can be higher than GPU kernel execution time. Finally, the overhead of the framework is considerable across all the workloads. It represents 95% of the execution time when running LeNet-5 on a CPU-GPU tandem with a single image. For the remaining workloads executing on a CPU-GPU tandem, it represents more than half of the execution time. The only exception being the ResNet-50 with a batch size of 128 images, where the framework overhead is reduced to 23% of the total execution time. Eager execution has to schedule every operation one-by-one. Hence, for the latency of the framework to be negligible for one operation, its enqueuing time has to be significantly shorter than the execution time of its kernel.

Fig. 4b, 5b, and 6b show the execution time per item, for LeNet-5, ResNet-50, and BERT, respectively. We observe that the execution time per item drastically reduces as the batch size increases. For example, the execution time per sentence of BERT becomes $37\times$ and $2019\times$ faster when increasing the batch size from 1 to 2048, running on a CPU-only and CPU-GPU tandem, respectively.

C. Time distribution between eager execution phases

We evaluate the distribution of the execution times of the different execution phases in the eager execution of all the operations of each workload.

Fig. 7a, 7b, and 7c show the distribution of the operations execution time between enqueuing time, dequeuing time, GPU execution time and CPU execution time for LeNet-5, ResNet-50, and BERT, respectively. Each distribution includes a marker to indicate the minimum, mean, and maximum values. We make three observations. First, the enqueuing, dequeuing and CPU execution times are largely independent from the batch size. We observe that the average enqueuing latency is around $0.1ms$ for each of our workloads. Although increasing the batch size induces larger kernels to run, this does not have an effect on the time needed to enqueue and dequeue the operations to and from the scheduling queue. Moreover, this batch size increase does not affect CPU kernel execution time. GPUs are more optimized to run highly dimensional computations. Hence, TensorFlow seems to map to GPU kernels most of the operations whose computational complexity is affected by the batch size. Second, the GPU execution time increases with the batch size. LeNet-5 goes from an average of $6\mu s$ to $180\mu s$ of GPU execution time when increasing the batch size from 1 to 8192 images. The average GPU execution time of ResNet-50 goes from $12\mu s$ to $500\mu s$ when increasing the batch size from 1 to 128 images. Finally, BERT goes from an average of $6\mu s$ to $200\mu s$ when increasing the batch size from 1 to 2048 sentences. This supports our previous observations, as larger batch sizes induce larger kernels runs, which results in relatively more GPU execution time. Third, the enqueuing time is shorter than the dequeuing time, independently from the batch size. Throughout all the workloads, the average dequeuing time is around $2\times$ the average enqueuing time.



(a) LeNet-5

(b) ResNet-50

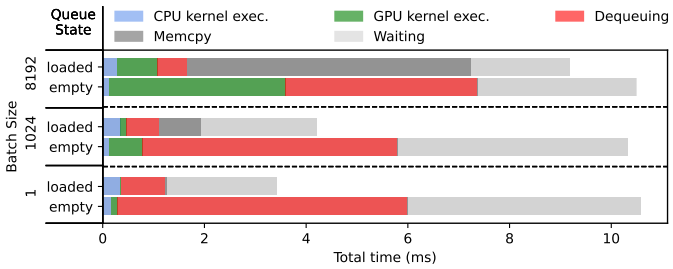
(c) BERT

Fig. 7: Inference on CPU-GPU tandem: distribution of TF eager execution time grouped by execution phase

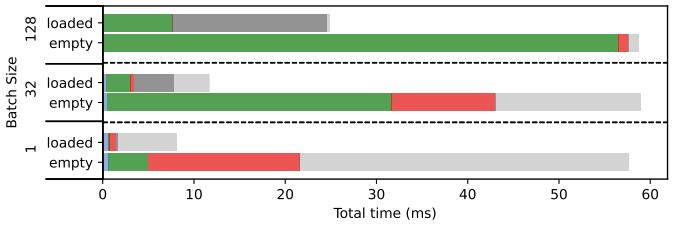
D. Utilization of the scheduling queue

We study the utilization of the scheduling queue and the corresponding execution time distribution across five categories: GPU kernel execution time, CPU kernel execution time, dequeuing time, memory transfer time, and waiting time (i.e., the rest of the execution). When several categories apply simultaneously, we select the category with the highest priority following the order of this list above. For example, we classify the concurrent time between GPU kernel execution and dequeuing as GPU kernel execution time.

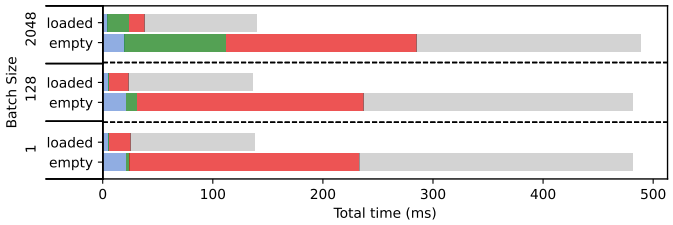
Fig. 8a, 8b and 8c show the execution time distribution with respect to the utilization of the scheduling queue. We make three observations. First, the utilization of the scheduling queue of the CNN workloads increases when increasing the batch size. The ResNet-50 goes from $8ms$ of queue utilization time to around $25ms$ ($3.1\times$) when increasing the batch size from 1 to 128. The LeNet-5 queue utilization also grows $3\times$ when increasing the batch size from 1 to 8192. In contrast, the utilization of the queue is minimally affected by the batch size in the case of BERT. Second, the scheduling queue is empty during most of the execution time. BERT exhibits an



(a) LeNet-5



(b) ResNet-50



(c) BERT

Fig. 8: Execution time distribution with respect to the utilization of the scheduling queue

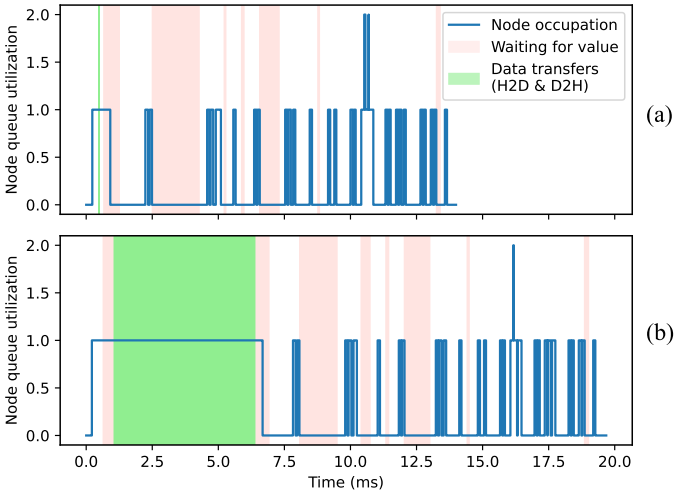


Fig. 9: LeNet-5 inference on a CPU-GPU tandem: utilization of the scheduling queue over time (a) with a single image, and (b) with batches of 8192 images

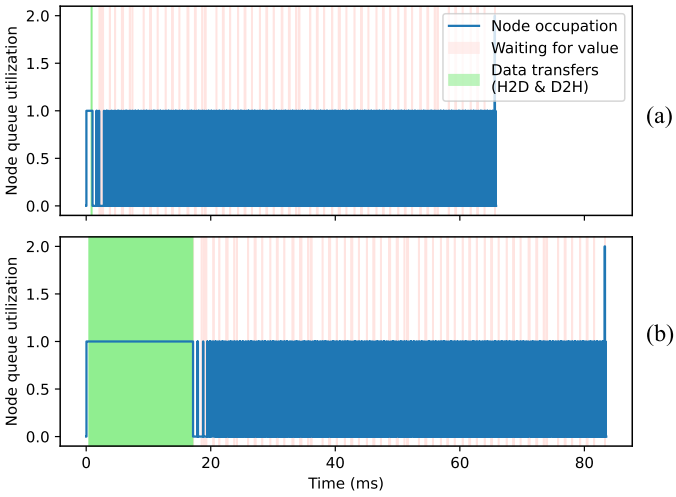


Fig. 10: ResNet-50 inference on a CPU-GPU tandem: utilization of the scheduling queue over time with a single image (a) and with batches of 128 images (b)

empty queue for 72% of its execution time in all batch sizes. ResNet-50 has an empty queue for 86% (58%) of the time for a batch size of 1 (128) images. Third, the waiting time represents a lower portion of the total execution time when increasing the batch size. BERT and ResNet-50 spend 10% and 31% less time waiting when increasing their batch sizes from 1 to 2048 and 128, respectively. However, we observe an exception with LeNet-5: when the queue is loaded, the time spent waiting for a node to be enqueued represents 64% for a batch size of 1 and 82% for a batch size of 8192 images.

To further understand the observed behaviors, we study the utilization of the scheduling queue over time. Fig. 9, 10 and 11 show the scheduling queue utilization for LeNet-5, ResNet-50 and BERT inference, respectively. We add annotations in the figures to indicate when the enqueuing thread is waiting for values to be computed and to show the data transfer timings. We make two additional observations. First, the LeNet-5 execution spends a considerable amount of time on data transfers, which can explain the previously described behavior of spending a lot of time waiting when the queue is loaded. Indeed, when subtracting the data transfer time from the waiting time, the remaining waiting time represents 64% with a batch size of 1 and only 28% with a batch size of 8192 images. Second, our workloads spend most of the time waiting for values to be computed. This limits the opportunities for the host program to advance enqueuing in parallel with kernel executions, limiting the amount of nodes enqueued. The zoomed part on Fig. 11 shows that the dequeuing thread empties the queue while the enqueuing thread waits for a value to be computed.

VI. Related works

To the best of our knowledge, this is the first paper providing a detailed description and analysis of the TensorFlow eager execution runtime. In this section, we identify two areas of related works that are relevant to our contributions.

On the one hand, there have been efforts around providing ML profiling platforms. However, these works are either dedicated to specific workloads [26] or evaluate the framework overhead without analyzing the execution of the

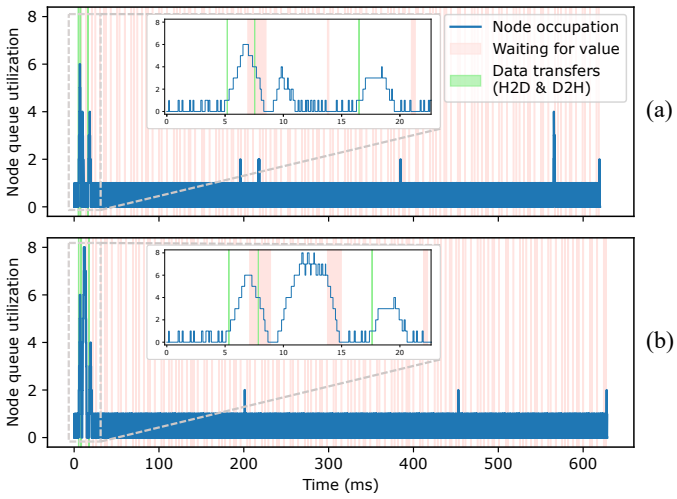


Fig. 11: BERT inference on a CPU-GPU tandem: utilization of the scheduling queue over time with a single sentence (a) and with batches of 2048 sentences (b)

runtime in detail [27] [28]. Our work is complementary to these contributions, as it provides a clear description of an ML framework mechanism. The new insights can be used on top of the one provided by current profiling platforms.

On the other hand, popular ML frameworks provide framework-specific profiling tools for developers to profile the performance of their model [21] [29] [30]. These profiling tools are intended for developers to help their model optimization. However, these tools combine application-level analysis with low-level information, which is often difficult to interpret by an average user. They do not include an accessible analysis of the performance of the inner mechanism of their specific frameworks. Our work aims to bridge this gap and provide new accessible insights on the TensorFlow framework runtime.

VII. Conclusion

We provide a detailed description of the main steps followed by the TensorFlow eager execution runtime to run code on a CPU-GPU tandem. We propose new metrics to analyze the ML framework's runtime performance overhead. We use our described approach to conduct in-depth profiling of the inference process of two CNNs (LeNet-5 and ResNet-50) and a transformer (BERT) for different batch sizes. Our results show that the runtime overhead of the ML framework is reduced considerably when operating with larger CPU and GPU kernels. However, we also show that the overhead could become significant when GPU kernel execution is not long enough to hide the framework's runtime latency. We believe that this work highlights the need to better understand ML framework's bottlenecks. Thus, we open source our profiler [14] and invite the community to build on our findings.

VIII. Acknowledgement

This work was performed using HPC/AI resources from GENCI-IDRIS (Grant 2022-AD011012967) and has been partially funded by the AdequatedDL (ANR-18-CE23-0012) and the F3CAS (ANR-20-CE25-0010) projects.

References

[1] Y. LeCun, B. Boser *et al.*, "Backpropagation applied to handwritten ZIP code recognition," *Neural Computation*, 1989.
[2] K. He, X. Zhang *et al.*, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.
[3] J. Devlin, M.-W. Chang *et al.*, "BERT: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2019.
[4] "An in-depth look at Google's first tensor processing unit." [Online]. Available: https://cloud.google.com/blog/products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu
[5] Y. Chen, T. Luo *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the International Symposium on Microarchitecture*, 2014.
[6] Y.-H. Chen, T.-J. Yang *et al.*, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2019.
[7] M. Abadi, P. Barham *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, 2016.
[8] A. Paszke, S. Gross *et al.*, "PyTorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, 2019.
[9] T. Chen, M. Li *et al.*, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.
[10] R. Elshawi, A. Wahab *et al.*, "DLBench: a comprehensive experimental evaluation of deep learning frameworks," *Cluster Computing*, 2021.
[11] A. Agrawal, A. Modi *et al.*, "TensorFlow eager: A multi-stage, python-embedded dsl for machine learning," in *Proceedings of Machine Learning and Systems*, 2019.
[12] R. M. Larsen and T. Shpeisman, "TensorFlow graph optimizations," 2019.
[13] "TensorFlow, large-scale machine learning on heterogeneous systems." [Online]. Available: https://github.com/tensorflow/tensorflow
[14] "TensorFlow eager runtime profiler repository." [Online]. Available: https://gite.lirmm.fr/adac/tensorflow-eager-runtime-profiler
[15] D. Kirk, "NVIDIA CUDA software and GPU parallel computing architecture," in *Proceedings of the International Symposium on Memory Management*, 2007.
[16] "Cuda, release: 10.2.89." [Online]. Available: https://developer.nvidia.com/cuda-toolkit
[17] "All symbols in TensorFlow 2." [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/all_symbols
[18] "TensorFlow guide: Create an op." [Online]. Available: https://www.tensorflow.org/guide/create_op
[19] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," 2010. [Online]. Available: http://eigen.tuxfamily.org
[20] S. Chetlur, C. Woolley *et al.*, "cuDNN: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.
[21] "TensorFlow profiler: Profile model performance." [Online]. Available: https://www.tensorflow.org/tensorboard/tensorboard_profiling_keras
[22] "NVIDIA CUPTI." [Online]. Available: https://docs.nvidia.com/cupti
[23] "Tensorboard.dev." [Online]. Available: https://tensorboard.dev/
[24] "Tensorflow Keras applications: Resnet50." [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/applications/resnet50
[25] "BERT base model (uncased)." [Online]. Available: https://huggingface.co/bert-base-uncased
[26] J. Gleeson, M. Gabel *et al.*, "RL-Scope: Cross-stack profiling for deep reinforcement learning workloads," in *Proceedings of Machine Learning and Systems*, 2021.
[27] C. Li, A. Dakkak *et al.*, "XSP: Across-stack profiling and analysis of machine learning models on GPUs," in *Proceedings of the International Parallel and Distributed Processing Symposium*, 2020.
[28] C. Li, A. Dakkak *et al.*, "The design and implementation of a scalable deep learning benchmarking platform," in *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, 2020.
[29] "PyTorch profiler." [Online]. Available: https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html
[30] "MXNet documentation: Profiling MXNet models." [Online]. Available: https://mxnet.apache.org/versions/master/api/python/docs/tutorials/performance/backend/profiler.html