Whisper JAX vs PyTorch: Uncovering the Truth about ASR Performance on GPUs

Deep Dive into Automatic Speech Recognition: Benchmarking Whisper JAX and PyTorch Implementation
s Across Platforms

Luís Roque
Towards Data Science

Luís Roque
Published in

Towards Data Science
·
8 min read
·
Apr 29

Introduction

In the world of Automatic Speech Recognition (ASR), speed and accuracy are of great importance. The si
ze of the data and models has been growing substantially recently, making it hard to be efficient. Nonethel
ess, the race is just starting, and we see new developments every week. In this article, we focus on Whisp
er JAX, a recent implementation of Whisper using a different backend framework that seems to run 70 tim
es faster than OpenAI's PyTorch implementation. We tested both CPU and GPU implementations and me
asured accuracy and execution time. Also, we defined experiments for small and large-size models while
parametrizing batch size and data types to see if we could improve it further.

As we saw in our previous article, Whisper is a versatile speech recognition model that excels in multiple
speech-processing tasks. It can perform multilingual speech recognition, translation, and even voice activi
ty detection. It uses a Transformer sequence-to-sequence architecture to predict words and tasks jointly.
Whisper works as a meta-model for speech-processing tasks. One of the downsides of Whisper is its effic
iency; it is often found to be fairly slow compared to other state-of-the-art models.

In the following sections, we go through the details of what changed with this new approach. We compare
Whisper and Whisper JAX, highlight the main differences between PyTorch and JAX, and develop a pipel
ine to evaluate the speed and accuracy between both implementations.
Figure 1: Can we make sense of sound efficiently? (source)

This article belongs to "Large Language Models Chronicles: Navigating the NLP Frontier", a new weekly s
eries of articles that will explore how to leverage the power of large models for various NLP tasks. By divin
g into these cutting-edge technologies, we aim to empower developers, researchers, and enthusiasts to h
arness the potential of NLP and unlock new possibilities.

Articles published so far:

- Summarizing the latest Spotify releases with ChatGPT
- Master Semantic Search at Scale: Index Millions of Documents with Lightning-Fast Inference Times usi
ng FAISS and Sentence Transformers
- Unlock the Power of Audio Data: Advanced Transcription and Diarization with Whisper, WhisperX, and
PyAnnotate

As always, the code is available on my Github.

PyTorch vs. JAX

The Machine Learning community extensively uses powerful libraries like PyTorch and JAX. While they s
hare some similarities, their inner works are quite different. Let's understand the main differences.

The AI Research Lab at Meta developed PyTorch and actively maintains it today. It is an open-source library based on the Torch library. Researchers widely use PyTorch due to its dynamic computation graph, intuitive interface, and solid debugging capabilities. The fact that it uses dynamic graphs gives it greater flexibility in building new models and simplifying the modification of such models during runtime. It is closer to Python and specifically to the NumPy API. The main difference is that we are not working with arrays but with tensors, which can run on GPU, and supports auto differentiation.

JAX is a high-performance library developed by Google. Conversely to PyTorch, JAX combines the benefits of static and dynamic computation graphs. It does this through its just-in-time compilation feature, which gives flexibility and performance. We can think of JAX being a stack of interpreters that progressively rewrite your program. It eventually offloads the actual computation to XLA — the Accelerated Linear Algebra compiler, also designed and developed by Google, to accelerate Machine Learning computations.
Building the ARS System that uses PyTorch or JAX

Let’s start by building a class to handle audio transcriptions using Whisper with PyTorch (OpenAI’s implementation) or Whisper with JAX. Our class is a wrapper for the models and an interface to easily set up experiments. We want to perform several experiments, including specifying the device, model type, and additional hyperparameters for Whisper JAX. Note that we used a singleton pattern to ensure that as we run several experiences, we do not end up with several instances of the model consuming our memory.

```
class Transcription:
    """
    A class to handle audio transcriptions using either the Whisper or Whisper JAX model.

    Attributes:
        audio_file_path (str): Path to the audio file to transcribe.
        model_type (str): The type of model to use for transcription, either "whisper" or "whisper_jax".
        device (str): The device to use for inference (e.g., "cpu" or "cuda").
        model_name (str): The specific model to use (e.g., "base", "medium", "large", or "large-v2").
        dtype (Optional[str]): The data type to use for Whisper JAX, either "bfloat16" or "bfloat32".
        batch_size (Optional[int]): The batch size to use for Whisper JAX.
    """
    _instance = None

    def __new__(cls, *args, **kwargs):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

    def __init__(
        self,
        audio_file_path: str,
        model_type: str = "whisper",
        device: str = "cpu",
        model_name: str = "base",
        dtype: Optional[str] = None,
        batch_size: Optional[int] = None,
    ):
        self.audio_file_path = audio_file_path
        self.device = device
        self.model_type = model_type
        self.model_name = model_name
        self.dtype = dtype
        self.batch_size = batch_size
        self.pipeline = None
```

The set_pipeline method sets up the pipeline for the specified model type. Depending on the value of the model_type attribute, the method initializes the pipeline using either by instantiating the FlaxWhisperPipline class for Whisper JAX or by calling the whisper.load_model() function for the PyTorch implementation of Whisper.

```
    def set_pipeline(self) -> None:
        """
        Set up the pipeline for the specified model type.

        Returns:
            None
        """
        if self.model_type == "whisper_jax":
            pipeline_kwargs = {}
            if self.dtype:
                pipeline_kwargs["dtype"] = getattr(jnp, self.dtype)
            if self.batch_size:
                pipeline_kwargs["batch_size"] = self.batch_size

            self.pipeline = FlaxWhisperPipline(
                f"openai/whisper-{self.model_name}", **pipeline_kwargs
            )
        elif self.model_type == "whisper":
            self.pipeline = whisper.load_model(
                self.model_name,
                torch.device("cuda:0") if self.device == "gpu" else self.device,
            )
        else:
            raise ValueError(f"Invalid model type: {self.model_type}")
```

The run_pipeline method transcribes the audio file and returns the results as a list of dictionaries containing the transcribed text and timestamps. In the case of Whisper JAX, it considers optional parameters like data type and batch size, if provided. Notice that you can set return_timestampsto False if you are only interested in getting the transcription. The model output is different if we run the transcription process with the PyTorch implementation. Thus, we must create a new object that aligns both return objects.

```
    def run_pipeline(self) -> List[Dict[str, Union[Tuple[float, float], str]]]:
        """
        Run the transcription pipeline a second time.

        Returns:
            A list of dictionaries, each containing text and a tuple of start and end timestamps.
        """
        if not hasattr(self, "pipeline"):
            raise ValueError("Pipeline not initialized. Call set_pipeline() first.")

        if self.model_type == "whisper_jax":
            outputs = self.pipeline(
                self.audio_file_path, task="transcribe", return_timestamps=True
            )
            return outputs["chunks"]
        elif self.model_type == "whisper":
            result = self.pipeline.transcribe(self.audio_file_path)
            formatted_result = [
                {
                    "timestamp": (segment["start"], segment["end"]),
                    "text": segment["text"],
                }
                for segment in result["segments"]
            ]
            return formatted_result
        else:
            raise ValueError(f"Invalid model type: {self.model_type}")
```

Finally, the transcribe_multiple() method enables the transcription of multiple audio files. It takes a list of a udio file paths and returns a list of transcriptions for each audio file, where each transcription is a list of dic tionaries containing text and a tuple of start and end timestamps.

```
def transcribe_multiple(
    self, audio_file_paths: List[str]
) -> List[List[Dict[str, Union[Tuple[float, float], str]]]]:
    """
    Transcribe multiple audio files using the specified model type.

    Args:
        audio_file_paths (List[str]): A list of audio file paths to transcribe.

    Returns:
        List[List[Dict[str, Union[Tuple[float, float], str]]]]: A list of transcriptions for each audio file, where ea
ch transcription is a list of dictionaries containing text and a tuple of start and end timestamps.
    """
    transcriptions = []

    for audio_file_path in audio_file_paths:
        self.audio_file_path = audio_file_path
        self.set_pipeline()
        transcription = self.run_pipeline()

        transcriptions.append(transcription)

    return transcriptions
```

Whisper JAX vs. PyTorch Performance Comparison
Experimental Setup

We used a long audio clip with more than 30 minutes to evaluate the performance of Whisper variants, wit h a PyTorch and JAX implementation. The researchers that developed Whisper JAX claim that the differe nce is more significant when transcribing long audio files.

Our experimental hardware setup consists of the following key components. For the CPU, we have an x8 6_64 architecture with a total of 112 cores, powered by an Intel(R) Xeon(R) Gold 6258R CPU running at 2 .70GHz. Regarding GPU, we use an NVIDIA Quadro RTX 8000 with 48 GB of VRAM.
Results and Discussion

In this section, we discuss the results obtained from the experiments to compare the performance of Whis per JAX and PyTorch implementations. Our results provide insights into the speed and efficiency of the tw o implementations on both GPU and CPU platforms.

Our first experiment involved running a long audio (over 30 minutes) using GPU and the larger Whisper m

odel (large-v2) that requires approximately 10GB of VRAM. Contrary to the claim made by the authors of Whisper JAX, our results indicate that the JAX implementation is slower than the PyTorch version. Even with the incorporation of half-precision and batching, we could not surpass the performance of the PyTorc h implementation using Whisper JAX. Whisper JAX took almost twice the time compared to the PyTorch i mplementation to perform a similar transcription. We also observed an unusually long transcription time w hen both half-precision and batching were employed.
Figure 2: Transcription execution time using Whisper's PyTorch implementation against Whisper JAX in G PU for the large model (image by author)

On the other hand, when comparing the CPU performance, our results show that Whisper JAX outperform s the PyTorch implementation. The speedup factor was approximately two times faster for Whisper JAX c ompared to the PyTorch version. We observed this pattern for the base and significant model variations.
Figure 3: Transcription execution time using Whisper's PyTorch implementation against Whisper JAX for t he base and large model in CPU (image by author)

Regarding the claim made by the authors of Whisper JAX that the second transcription should be much fa ster, our experiments did not provide supporting evidence. The difference in speed between the first and s econd transcriptions was not significant. Plus, we found that the pattern was similar between both Whispe r and Whisper JAX implementations.
Conclusion

In this article, we presented a comprehensive analysis of the Whisper JAX implementation, comparing its performance to the original PyTorch implementation of Whisper. Our experiments aimed to evaluate the cl aimed 70x speed improvement using a variety of setups, including different hardware and hyperparameter s for the Whisper JAX model.

The results showed that Whisper JAX outperformed the PyTorch implementation on CPU platforms, with a speedup factor of approximately two fold. Nonetheless, our experiments did not support the authors' clai ms that Whisper JAX is significantly faster on GPU platforms. Actually, the PyTorch implementation perfor med better when transcribing long audio files using a GPU.

Additionally, we found no significant difference in the speed between the first and second transcriptions, a claim made by the Whisper JAX authors. Both implementations exhibited a similar pattern in this regard.

Keep in touch: LinkedIn