# A Simple Guide to Enabling CUDA GPU Support for llama-cpp-python on Your OS or in Containers

<https://medium.com/@ryan.stewart113/a-simple-guide-to-enabling-cuda-gpu-support-for-llama-cpp-python-on-your-os-or-in-containers-8b5ec1f912a4>

A cartoon snake in a room

Description automatically generated

A GPU can significantly speed up the process of training or using large-language models, but it can be challenging just getting an environment set up to use a GPU for training or inference — there are plenty of conversations on StackOverflow and other developer forums to attest to this.

If you want to learn how to enable the popular llama-cpp-python library to use your machine’s CUDA-capable GPU, you’ve come to the right place. Fortunately it is a very straightforward process once you filter throught the noise, which is exactly what I’ve done for you here. In this guide, I‘ll walk you through the specific steps required to enable GPU support for llama-cpp-python.

(The steps below assume you have a working python installation and are at least familiar with llama-cpp-python or already have llama-cpp-python working for CPU only).

# Step 1: Download & Install the CUDA Toolkit

The first step in enabling GPU support for llama-cpp-python is to download and install the NVIDIA CUDA Toolkit. The CUDA Toolkit includes the drivers and software development kit (SDK) required to compile and run CUDA-accelerated applications.

You can download the CUDA Toolkit installer and find installation instructions for Windows and various, popular Linux distributions on Nvidia’s website: <https://developer.nvidia.com/cuda-downloads>.

# Step 2: Use CUDA Toolkit to Recompile llama-cpp-python with CUDA Support

Once you have installed the CUDA Toolkit, the next step is to compile (or recompile) llama-cpp-python with CUDA support enabled. This involves setting the appropriate environment variables to point to your nvcc installation (which is installed with the CUDA Toolkit) and specifying the CUDA architecture(s) to compile for.

Here’s an example command to recompile llama-cpp-python with CUDA support enabled for all major CUDA architectures:

CUDACXX=/usr/local/cuda-12/bin/nvcc CMAKE\_ARGS="-DLLAMA\_CUBLAS=on -DCMAKE\_CUDA\_ARCHITECTURES=all-major" FORCE\_CMAKE=1 pip install llama-cpp-python --no-cache-dir --force-reinstall --upgrade

In the example above, we are setting the CUDACXX environment variable to the path of the nvcc compiler executable included with the CUDA Toolkit. We are also setting the CMAKE\_ARGS variable to specify that we want to enable CUDA support (-DLLAMA\_CUBLAS=on) and compile for all major CUDA architectures (-DCMAKE\_CUDA\_ARCHITECTURES=all-major).

Other valid values for CMAKE\_CUDA\_ARCHITECTURES are all (for all) or native to build for the host system's GPU architecture. Note that GPUs are usually not available while building a container image, so avoid using -DCMAKE\_CUDA\_ARCHITECTURES=native in a Dockerfile unless you know what you're doing.

# Example Dockerfile

To make it easier to run llama-cpp-python with CUDA support and deploy applications that rely on it, you can build a Docker image that includes the necessary compile-time and runtime dependencies. Below I’ve provided a Dockerfile demonstrating the steps explained above; the Dockerfile builds a Docker image with llama-cpp-python compiled with CUDA support for all major CUDA architectures:

FROM python:3.10-bookworm  
  
## Add your own requirements.txt if desired and uncomment the two lines below  
# COPY ./requirements.txt .  
# RUN pip install -r requirements.txt  
  
## Install CUDA Toolkit (Includes drivers and SDK needed for building llama-cpp-python with CUDA support)  
RUN apt-get update && apt-get install -y software-properties-common && \  
 wget https://developer.download.nvidia.com/compute/cuda/12.3.1/local\_installers/cuda-repo-debian12-12-3-local\_12.3.1-545.23.08-1\_amd64.deb && \  
 dpkg -i cuda-repo-debian12-12-3-local\_12.3.1-545.23.08-1\_amd64.deb && \  
 cp /var/cuda-repo-debian12-12-3-local/cuda-\*-keyring.gpg /usr/share/keyrings/ && \  
 add-apt-repository contrib && \  
 apt-get update && \  
 apt-get -y install cuda-toolkit-12-3   
  
## Install llama-cpp-python with CUDA Support (and jupyterlab)  
RUN CUDACXX=/usr/local/cuda-12/bin/nvcc CMAKE\_ARGS="-DLLAMA\_CUBLAS=on -DCMAKE\_CUDA\_ARCHITECTURES=all-major" FORCE\_CMAKE=1 \  
 pip install jupyterlab llama-cpp-python --no-cache-dir --force-reinstall --upgrade  
  
WORKDIR /workspace  
  
## Run jupyterlab on container startup  
CMD ["jupyter", "lab", "--ip", "0.0.0.0", "--port", "8888", "--NotebookApp.token=''", "--NotebookApp.password=''", "--no-browser", "--allow-root"]

In this example, we use a Debian-based Python 3.10 image as our base image. We then install the CUDA Toolkit and compile and install llama-cpp-python with CUDA support (along with jupyterlab). Finally, we set our container’s default command to run JupyterLab when the container starts.

The Dockerfile above can be used as-is or modified to suit your needs. If you are using a container to run your llama-cpp-python projects, don’t forget to [ensure that your GPU is available to your containers at runtime](https://docs.docker.com/config/containers/resource_constraints/#gpu)! Otherwise, compiling with CUDA support will be for naught.

# Conclusion

At this point, you should be set up to use llama-cpp-python with GPU on your host operating system or in containers. If you’re wondering what to do next, try downloading and using some GGUF models with llama-cpp-python as I’ve explained here — <https://stackoverflow.com/questions/77630013/how-to-run-any-gguf-model-using-transformers-or-any-other-library/77734862#77734862>.

Let me know what kinds of cool projects you build using llama-cpp-python in the comments below. Happy New Year, and happy coding!

# A Simple, Practical Guide to Running Large-Language Models on Your Laptop

<https://medium.com/predict/a-simple-comprehensive-guide-to-running-large-language-models-locally-on-cpu-and-or-gpu-using-c0c2a8483eee>

Deploying your large-language models (LLMs), either “as-a-service” or self-managed, can help reduce costs and improve operations and scalability (and are almost always a must for production use). However, there are plenty of great reasons to run models locally (i.e. on your laptop or desktop). Rapid prototyping & development, learning, and experimenting with large-language models (LLMs) are just a few activities that can benefit from the ability to run models locally (not to mention the personal-data-privacy benefits of running models locally). Since running models locally can both reduce the cost and increase the speed with which you can iterate on your LLM-powered apps, being able to run local models can even have a positive, tangible impact on the quality of your production applications.

Almost all of us are familiar with AI and LLMs and have interacted with LLMs through online services, but it’s safe to say that a majority of the population (maybe even when just considering the “technical” population”) has never realized just how easily they can run models on their own computer. I have seen a variation of the question, “How can I run a large-language model on my laptop?”, hundreds of times on forums, so I’m going to show you exacctly how to do that, step-by-step.

In this guide, I will walk you through the process of downloading a GGUF model-fiLE from HuggingFace Model Hub, installing llama-cpp-python,and running the model on CPU (and/or GPU). If the terms “GGUF” or “llama-cpp-python” are new to you, the following background may be helpful. However, if you are just looking for code, feel free to jump down to the “Detailed Steps” section below.

# Background

## What are GGUF models and where do I get them?

GGUF is a flexible, extensible, “future-proof” file format for storing, sharing, and loading quantized LLMs that can run on both CPU and GPU (or both with layer-offloading). A detailed discussion about quantization and other quantized-model formats (e.g. GPTQ or AWQ) is beyond the scope of this guide. Suffice it to say that quantization essentially allows us to compress the parameters that make up the weights-matrices of an LLM (thereby compressing the LLM itself) so that we can effectively run LLMs that would otherwise be too resource-intensive for our machines.

For example, if we have an original model with 13-billion parameters and each parameter takes up two (2) bytes (i.e. 16 bits), then the model will require approximately 13-billion \* 2 = 26-billion bytes (or ~26GB) of RAM to load (plus a little more when we start passing tokens through it). However, if we take that model and quantize it so that each parameter only takes up half (0.5) of a byte (i.e. 4 bits), we can now load that model with quarter of the original RAM (or ~6.5 GB). Reducing the model size doesn’t just help us load it into memory — it also speeds up inference and reduces both the time it takes to transfer the model across a network and the space it takes on disk.

Our altruistic friend Tom Jobbins (better known as “TheBloke”) takes care of the heavy lifting of quantization for us, so all we have to do is download the already-quantized models from [his GGUF repositories on HuggingFace Model Hub](https://huggingface.co/TheBloke?search_models=GGUF&sort_models=downloads#models).

While the specific models that you can run will vary depending on your machine’s capabilities, there are some great models out there that can generate text at a reasonable speed and with impressive context windows on anywhere from 4–8GB RAM thanks to quantization (I’m looking at you [Mistral-7b](https://huggingface.co/TheBloke/Mistral-7B-Instruct-v0.1-GGUF) and [Vicuna-13b](https://huggingface.co/TheBloke/vicuna-13B-v1.5-16K-GGUF)). And with ~25GB RAM you can even run [Mixtral-8x7b](https://huggingface.co/TheBloke/Mixtral-8x7B-Instruct-v0.1-GGUF), which [outperforms both GPT-3.5 and Llama-70b on most benchmarks](https://mistral.ai/news/mixtral-of-experts/).

## What is llama-cpp-python?

Llama-cpp-python is a python binding (or adapter) for llama.cpp, which is a library that was originally developed at Facebook (i.e. Meta) for using pure C/C++ to run quantized models, including GGUF models, at speed (with optional GPU support). Llama-cpp-python makes it easy for those who know Python but don’t know C/C++ to benefit from the speed and performance offered by llama.cpp.

For those of you using GPUs, llama.cpp (and by extension llama-cpp-python) is particularly helpful because it gracefully handles the complex task of offloading a portion of a model’s layers onto GPU. This means you can benefit from GPU acceleration even when using models that can’t fit entirely in VRAM.

Pre-quantized GGUF models and llama-cpp-python make a potent combination, because they allow us to quickly and easily run powerful large-language models on our regular consumer hardware. Below I’ll show you exactly how to do that.

# Detailed Steps (with Code)

The steps below assume you have a working Python3 installation and that the pip command points to the correct pip.

## Install Pre-Requisites

To install llama-cpp-python, run the following command in your terminal:

pip install llama-cpp-python

You will also need to install huggingface\_hub,which we will use to download GGUF models from HuggingFace Model Hub. Run the following command:

pip install huggingface\_hub

Now that we have our dependencies downloaded, we can start writing some Python to download and run some models.

## Choose and Download a Model

In the following code, we download the mixtral-8x7b-instruct-v0.1-gguf GGUF file (containing the model with 4-bit, "type-1" quantized weights which provide a healthy balance of compression and model performance; for more information about quantization methods, see [this](https://huggingface.co/TheBloke/Mixtral-8x7B-Instruct-v0.1-GGUF#explanation-of-quantisation-methods)) from the TheBloke/mixtral-8x7b-instruct-v0.1-gguf repository on HuggingFace Model Hub.

## Imports  
from huggingface\_hub import hf\_hub\_download  
  
## Define model name and file name  
model\_name = "TheBloke/Mixtral-8x7B-Instruct-v0.1-GGUF"  
model\_file = "mixtral-8x7b-instruct-v0.1.Q4\_K\_M.gguf"  
  
## Use the following model\_name and model\_file if you have 8gb ram or less  
# model\_name = "TheBloke/Mistral-7B-OpenOrca-GGUF"  
# model\_file = "mistral-7b-openorca.Q4\_K\_M.gguf"  
  
## Use the following model\_name and model\_file if you have 16gb ram or less  
# model\_name = "TTheBloke/vicuna-13B-v1.5-16K-GGUF"  
# model\_file = "vicuna-13b-v1.5-16k.Q4\_K\_M.gguf"  
  
## Download the model  
model\_path = hf\_hub\_download(model\_name, filename=model\_file)

I have also provided model\_names and model\_files for smaller models in case you are unable or just don’t want to run the mixtral-8x7b GGUF model — simply comment out the lines containing the mixtral name and file and un-comment the lines for the model you want to use.

You are also free to choose any other model you please, although as mentioned above I highly recommend using one of [TheBloke’s GGUF models](https://huggingface.co/TheBloke?search_models=GGUF&sort_models=downloads#models). You will need to copy both the model name (also referred to as the “repo” or “repo ID”) and the specific file from the repo that you want to use. Click the “Files” tab in a HuggingFace Model Hub GGUF model repo to see the available quantized model files.

## Load the Model

Now that you have the model downloaded, you can use it by instantiating the model object and passing in the path to the downloaded model file and some optional model parameters as keyword-arguments as shown below:

## Imports  
from llama\_cpp import Llama  
  
model\_kwargs = {  
 "n\_ctx":4096, # Context length to use  
 "n\_threads":4, # Number of CPU threads to use  
 "n\_gpu\_layers":0,# Number of model layers to offload to GPU. Set to 0 if only using CPU  
}  
  
## Instantiate model from downloaded file  
llm = Llama(model\_path=model\_path, \*\*model\_kwargs)

I have provided some sensible defaults that will work for CPU-only inference in the code above, but you can find an exhaustive list of model parameters for llama-cpp-python [here](https://llama-cpp-python.readthedocs.io/en/latest/api-reference/).

Note that if you are trying to use a CUDA-capable GPU, you will need to compile and install (or reinstall) llama-cpp-python with CUDA support as explained in my article, “[A Simple Guide to Enabling CUDA GPU Support for llama-cpp-python on Your OS or in Containers](https://medium.com/@ryan.stewart113/a-simple-guide-to-enabling-cuda-gpu-support-for-llama-cpp-python-on-your-os-or-in-containers-8b5ec1f912a4)”.

## Run Inference with the Model

You can pass in several parameters to control the text-generation process, such as the maximum number of tokens to generate, stopping criteria, and the top\_k (i.e. number of most-likely tokens to consider at each generation step). We run inference (i.e. generate text) using our LLM by passing in a prompt and the generation parameters.

## Generation kwargs  
generation\_kwargs = {  
 "max\_tokens":200, # Max number of new tokens to generate  
 "stop":["<|endoftext|>", "</s>"], # Text sequences to stop generation on  
 "echo":False, # Echo the prompt in the output  
 "top\_k":1 # This is essentially greedy decoding, since the model will always return the highest-probability token. Set this value > 1 for sampling decoding  
}  
  
## Run inference  
prompt = "The meaning of life is "  
res = llm(prompt, \*\*generation\_kwargs) # Res is a dictionary  
  
## Unpack and the generated text from the LLM response dictionary and print it  
print(res["choices"][0]["text"])

I have provided some sensible defaults, but you can find an exhaustive list of generation parameters [here](https://llama-cpp-python.readthedocs.io/en/latest/api-reference/).

# Full Code

Here’s the full code collated from the snippets above:

## Imports  
from huggingface\_hub import hf\_hub\_download  
from llama\_cpp import Llama  
  
## Define model name and file name  
model\_name = "TheBloke/Mixtral-8x7B-Instruct-v0.1-GGUF"  
model\_file = "mixtral-8x7b-instruct-v0.1.Q4\_K\_M.gguf"  
  
## Use the following model\_name and model\_file if you have 8gb ram or less  
# model\_name = "TheBloke/Mistral-7B-OpenOrca-GGUF"  
# model\_file = "mistral-7b-openorca.Q4\_K\_M.gguf"  
  
## Use the following model\_name and model\_file if you have 16gb ram or less  
# model\_name = "TTheBloke/vicuna-13B-v1.5-16K-GGUF"  
# model\_file = "vicuna-13b-v1.5-16k.Q4\_K\_M.gguf"  
  
## Download the model  
model\_path = hf\_hub\_download(model\_name, filename=model\_file)  
  
  
model\_kwargs = {  
 "n\_ctx":4096, # Context length to use  
 "n\_threads":4, # Number of CPU threads to use  
 "n\_gpu\_layers":0,# Number of model layers to offload to GPU. Set to 0 if only using CPU  
}  
  
## Instantiate model from downloaded file  
llm = Llama(model\_path=model\_path, \*\*model\_kwargs)  
  
## Generation kwargs  
generation\_kwargs = {  
 "max\_tokens":200, # Max number of new tokens to generate  
 "stop":["<|endoftext|>", "</s>"], # Text sequences to stop generation on  
 "echo":False, # Echo the prompt in the output  
 "top\_k":1 # This is essentially greedy decoding, since the model will always return the highest-probability token. Set this value > 1 for sampling decoding  
}  
  
## Run inference  
prompt = "The meaning of life is "  
res = llm(prompt, \*\*generation\_kwargs) # Res is a dictionary  
  
## Unpack and the generated text from the LLM response dictionary and print it  
print(res["choices"][0]["text"])

# Conclusion

At this point, you should be off to the races. If you’re wondering what to do next, consider playing around with different models and prompts or start building a chatbot or RAG application around your locally-running LLMs.

If you need some inspiration, take a look at [my generative-AI GitHub repo](https://github.com/rylativity/generative-ai), which includes Python notebooks and a Streamlit app with a user-interface for playing with LLMs for various use cases.

Let me know what kinds of cool projects you build in the comments below. Happy New Year, and happy coding!