GPU Hardware Acceleration with Tensorflow using WSL2 on Windows 11

Audience:

This guide is for data scientists and ML/AI engineers with Windows 11 laptops with NVIDIA GPUs installed who want to leverage the GPU for processing workloads developed in Python environments directly on their laptops.

Purpose:

This document provides a step-by-step guide for data scientists and ML/AI engineers who want to leverage an installed NVIDIA GPU on their laptop to run workloads for model development.

It is very difficult for someone new to ML/AI environment management to find definitive resources to successfully configure their local Windows 11 laptop to process GPU workloads using TensorFlow.

Alternatives:

Many cloud-based options are available for running workloads on cloud GPUs exist (i.e. Google Colab Pro).

Problem Statement:

TensorFlow versions 2.11 and newer no longer provide native, direct GPU support for Windows.

- Developers cannot simply pip install tensorflow and expect it to automatically leverage their NVIDIA GPU with CUDA on a native Windows 11 environment for newer TensorFlow versions.
- This change removes the convenience of a direct Windows installation and can introduce performance overhead or installation complexity compared to native Linux environments, where TensorFlow GPU support remains straightforward.

Solution:
The Windows Subsystem for Linux 2 acts as a bridge to use the Windows computer normally, but also enables a high-performance Linux environment to run TensorFlow software to fully use the NVIDIA graphics card without complications.

Using WSL2 creates a streamlined environment for GPU workloads.

- WSL2 sets up a lightweight, dedicated and efficient Linux operating system within your Windows machine.
- It "shares" your NVIDIA tools: Microsoft and NVIDIA worked together so that the powerful NVIDIA graphics card drivers you installed for Windows can be used by the Linux workshop.
- TensorFlow runs as fast and efficiently inside this WSL2 as it would on a dedicated Linux computer.

- **Additional Background on the WSL2 Environment:**
  - The Windows Subsystem for Linux 2 (WSL2) provides a full Linux kernel running as a lightweight virtual machine directly on Windows. This allows users to install and run standard Linux distributions (like Ubuntu) within Windows. Crucially, Microsoft and NVIDIA have collaborated to enable GPU passthrough to WSL2. This means the NVIDIA GPU drivers installed on the Windows *host* are accessible from within the Linux environment running in WSL2, allowing TensorFlow to leverage CUDA for GPU acceleration as if it were running on a native Linux system, with near-native performance.

- **Important Technical Information:**
  - TensorFlow 2.10 is the last version that officially supports GPU acceleration directly on native Windows.
  - Unlike PyTorch, where CUDA/cuDNN are often bundled or implicitly handled, TensorFlow on Windows natively requires a very specific setup of CUDA Toolkit and cuDNN libraries.
  - NVIDIA Studio Driver 576.80 is compatible with CUDA 11.2.

Web References:

- https://developer.nvidia.com/cuda-gpus
- https://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html#cuda-major-component-versions
- https://developer.nvidia.com/cuda-downloads?target_os=Linux&target_arch=x86_64&Distribution=WSL-Ubuntu&target_version=2.0&target_type=deb_local
- https://developer.download.nvidia.com/compute/cuda/repos/wsl-ubuntu/x86_64/

Glossary:

- Sudo = (Super User DO) command in Linux

Pre-Requisites:

- Use the NVIDIA App installed on your Windows 11 laptop to download and install the latest NVIDIA Studio Driver for your GPU.
- Check your NVIDIA GPU and Driver.
    - Open command prompt and run the following command: nvidia-smi
    - Note the following:
        - The driver version
        - The CUDA version.
            - This indicates the *maximum* CUDA version the *driver* supports
            - The NVIDIA driver is backward compatible with older CUDA versions.

My Specific Environment:

- Dell XPS 15 Laptop
- Windows 11 operating system (Windows 11 Home, OS Build 26100.4349, Windows Feature Experience Pack 1000.26100.107.0)
- NVIDIA GeForce RTX 4050 Laptop GPU
- NVIDIA Studio Driver version 576.80 (Compatible with CUDA version 12.9 and before).
- VS Code Version 1.101.2

Installing the WSL2 (for CUDA version 12.9 and prior given backwards compatibility:

1. Run PowerShell as an administrator
2. Enable the Virtual Machine Platform and the Windows Subsystem for Linux (WSL) by running these two commands in sequence:
    - dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /norestart
    - /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all /norestart
3. Install a WSL2 Linux Distribution (e.g., Ubuntu)
    - wsl –install
4. Enter a username (chris) and password (GPUuser)
5. Install the NVIDIA CUDA toolkit by executing the following one by one in sequence in the WSL2 Ubuntu terminal:
    - wget https://developer.download.nvidia.com/compute/cuda/repos/wsl-ubuntu/x86_64/cuda-wsl-ubuntu.pin
    - sudo mv cuda-wsl-ubuntu.pin /etc/apt/preferences.d/cuda-repository-pin-600
    - wget https://developer.download.nvidia.com/compute/cuda/12.9.1/local_installers/cuda-repo-wsl-ubuntu-12-9-local_12.9.1-1_amd64.deb

- sudo dpkg -i cuda-repo-wsl-ubuntu-12-9-local_12.9.1-1_amd64.deb
- sudo cp /var/cuda-repo-wsl-ubuntu-12-9-local/cuda-*-keyring.gpg /usr/share/keyrings/
- sudo apt-get update
- sudo apt-get -y install cuda-toolkit-12-9

6. Configure CUDA Environment Variables in WSL2 by executing the following one by one in sequence in the WSL2 Ubuntu terminal:
   - echo 'export PATH=/usr/local/cuda-12.9/bin${PATH:+:${PATH}}' >> ~/.bashrc
   - echo 'export LD_LIBRARY_PATH=/usr/local/cuda-12.9/lib64${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}' >> ~/.bfriashrc
   - source ~/.bashrc
7. Verification steps in the WSL2 Ubuntu terminal
   - nvcc --version
     i. Expected: version 12.9
   - echo $LD_LIBRARY_PATH
     i. Expected: /usr/local/cuda-12.9/lib64
8. Download and install miniconda for Linux by executing the following one by one in sequence in the WSL2 Ubuntu terminal:
   - wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
   - bash Miniconda3-latest-Linux-x86_64.sh
   - rm Miniconda3-latest-Linux-x86_64.sh
9. Close the Ubuntu terminal, and re-open a new one.
   - Run conda –version to validate miniconda was installed correctly.
10. Create and activate the Conda Environment for TensorFlow in WSL2
    - conda create --name my_TensorFlow_WSL_env python=3.10
    - conda activate my_TensorFlow_WSL_env
11. Validate the environment and python version
    - conda env list
    - python --version
12. Install TensorFlow in the active “my_TensorFlow_WSL_env” Python environment.
    - pip install tensorflow[and-cuda]
13. Install other typical data science libraries in the active “my_TensorFlow_WSL_env” Python environment.
    - conda install numpy pandas scikit-learn matplotlib jupyterlab Ipykernel
14. Install the remote development extension from Microsoft in VS Code.
    - This allows you to open any folder in a container - in this case the Windows Subsystem for Linux (WSL) and take advantage of VS Code’s full feature set.
15. Open your WSL2 Home Directory in VS Code:
    - Open the command Palette
    - Type WSL
    - Select Remote-WSL: New WSL Window
    - Alternative
      i. Open a WSL Ubuntu Terminal
      ii. Navigate to the directory you want to open in VS Code

    iii. Once in the directory, type “code .”

16. Install the VS Code Python extension in your WSL2 Environment
17. Go to the command palette and search for Python: Select Interpreter
    - Select your TensorFlow environment created previously.
18. Create TensorFlow Test File and Validate GPU Usage in VS Code
    - Right click on the explorer and add file named “tf_gpu_test.py”
    - Add the sample code as shown here

```
import tensorflow as tf
import time
import numpy as np
import pandas as pd
import sklearn
import matplotlib.pyplot as plt

# Enable device placement logging to see where ops run
# This is crucial for verifying GPU usage in the logs
tf.debugging.set_log_device_placement(True)

print("\n--- Starting WSL2 TensorFlow GPU Verification Test ---")

# 1. List physical GPUs
physical_gpus = tf.config.list_physical_devices('GPU')
print(f"Num Physical GPUs Available: {len(physical_gpus)}")
for gpu in physical_gpus:
    print(f"  {gpu}")

# 2. List logical GPUs (should appear if physical are found and
initialized)
logical_gpus = tf.config.list_logical_devices('GPU')
print(f"Num Logical GPUs Available: {len(logical_gpus)}")
for gpu in logical_gpus:
    print(f"  {gpu}")

# 3. Check if TensorFlow was built with CUDA support (should be True)
print(f"TensorFlow built with CUDA: {tf.test.is_built_with_cuda()}")

# 4. Perform a small operation explicitly on GPU and check its device
try:
    # tf.test.gpu_device_name() is a quick check, but actual device
placement logs are more definitive
```

```python
    device_name = tf.test.gpu_device_name()
    print(f"tf.test.gpu_device_name() reports: {device_name}")

    # Explicitly place a small operation on the GPU:0
    with tf.device('/GPU:0'): # Explicitly target GPU 0 in case it's not
default
        a = tf.constant([[1.0, 2.0], [3.0, 4.0]])
        b = tf.constant([[5.0, 6.0], [7.0, 8.0]])
        c = tf.matmul(a, b)
        print(f"Explicit GPU MatMul result: {c.numpy()}")
        # Check the device of the tensor after the operation
        print(f"Explicit GPU MatMul ran on device: {c.device}")


    # 5. Perform a slightly larger operation to observe GPU utilization
    # This part is more likely to show up on nvidia-smi
    matrix_size = 2048 # Adjust this size if you want more/less load
    A = tf.random.normal((matrix_size, matrix_size), dtype=tf.float32)
    B = tf.random.normal((matrix_size, matrix_size), dtype=tf.float32)

    start_time = time.time()
    with tf.device('/GPU:0'): # Again, explicitly targeting GPU 0
        C = tf.matmul(A, B)
        D = tf.nn.relu(C)
        E = tf.reduce_sum(D)
    end_time = time.time()

    print(f"\nLarger tensor 'E' is on device: {E.device}")
    print(f"Large matrix ops took {end_time - start_time:.4f} seconds.")
    print("TensorFlow GPU operations seem to be working!")

except Exception as e:
    print(f"\n--- GPU ERROR: {e} ---")
    print("TensorFlow failed to run operations on GPU in WSL2. Error
details:")
    print(e) # Print the actual error message

print("\n--- End of WSL2 TensorFlow GPU Verification Test ---")
```

19. Go to the Run and Debug icon on the left hand side and create a launch.json file.
    - In the command palette, select “Python Debugger” then “Python file.
    - VS code will generate the launch.json file. Edit the file to match the following and save the file.

```
{


    "version": "0.2.0",
    "configurations": [


        {
            "name": "Python: Current File (TensorFlow WSL2 Verbose GPU)",
            "type": "debugpy",
            "request": "launch",
            "program": "${file}",
            "console": "integratedTerminal",
            "justMyCode": true,
            "env": {
                "TF_CPP_MIN_LOG_LEVEL": "0"
            }
        }
    ]
}
```

- Select the “tf_gpu_test.py” file and make sure it is opened in VS code (click to open if it is not already).
- Go to run and debug and ensure the following is selected:
    i. Python: Current File (TensorFlow WSL2 Verbose GPU)
- Open up an Ubuntu terminal and use the following command to start monitoring the GPU usage:
    i. watch -n 1 nvidia-smi
- Open the task manager and watch the NVIDIA GPU
    i. Seeing a spike in the Task Manager Performance screen for your NVIDIA GPU during the execution of the TensorFlow script confirms that the computations are indeed being offloaded from your CPU to your GPU.