Steps to take

1. A VPN connection is needed to access CAIR through WiFi.
    1. Download CISCO AnyConnect VPN software
    2. Connect to this address: connect.cair.mun.ca
    3. The VPN is safe. If you faced a warning message, click "Change Setting" and then uncheck the "Block connections to untrusted servers"
    4. Set the VPN group to HPC
    5. Finally use your provided CAIR username and password to connect.

2. Uploading Files to the CAIR computing environment
    1. Download an STFP program such as WinSCP for Windows or FileZilla for Linux and MacOS
    2. Add a new site to your program with these specifications
        1. Protocol: STFP or SCP
        2. Host name: login1.cair.mun.ca or login2.cair.mun.ca
        3. Port number: 22
        4. Username and password: Your provided CAIR credentials
    3. Trust the server and add its host key to a cache if prompted.
    4. Now you can drag and drop files from your local device to CAIR
    5. You can also manage files on CAIR storage by running your scripts or using the command line if you wish. (accessing the command line is explained in the next section)

3. Accessing the command line
    - An Open SSH client is required.
        1. On windows, you can download and use PuTTY. Use configurations given below to connect properly. You will be prompted for your credentials in the command line.
           - Host name: login1.cair.mun.ca or login2.cair.mun.ca
           - Port: 22
           - Connection type: SSH

           You can then click “Open” to open the terminal and then enter your username and then password to enter.
        2. On Linux and MacOS you can use the terminal. Type the command: "**ssh USERNAME@login1.cair.mun.ca**" or the command "**ssh USERNAME@login2.cair.mun.ca**" to connect to CAIR but replace the USERNAME in the command with your CAIR username.You will be prompted for your password next.

4. You now have access to the command line. To run your code, you will need to load software into your $PATH using modules. You can use these commands for managing the modules:
    - **module avail** → Provides a list of available modules
    - **module load MODULE_NAME**
        - Loads a module to use. Replace the MODULE_NAME with the desired module name.
        - The PyTorch module named "*miniconda/pytorch-tensorflow-gpu-2024.09*" is pre-configured and can provide you with out of the box tools.
        - If your loaded module has missing packages, you can use the pip command with --user flag to install packages.
            - example: pip install --user transformers
                - This examples installs the "transformers" package provided by Hugging Face

    - **which conda** → Checks the path for Conda
    - **module list** →Provides a list of loaded modules
    - **module purge** → Clears all loaded modules once completed

5. There is a different queue for each type of compute available. Some of the types are "normal", "GPU", "large_memory", and legacy computes. You can run the command "bqueues" to get a list of queues and number of "jobs" in them.

6. To run code, create a (python) script file that contains the code. Then you can submit your code to be run as a "job". Commands for managing "jobs" are given below. Note that if the script or the output files are not in the current working directory, the path to the files should be provided. The current working directory can be found by entering the **pwd** command and the files in this directory can be listed with the **ls** command.
    - **bjobs -a**
        - Shows a list of all the previous jobs submitted and their status
        - Jobs that are completed are described with DONE, the ones that faced an error are described with EXIT, and the ones that are pending are described with PEND.

    - **bjobs -| JOB_ID**
        - Checks the status of a specific job with the specified JOB_ID.

    - **bsub**
        - Submits a job. There are many configurations available for this command and only the main ones are described here. Jobs can have different names, be submitted to different queues to use specific resources such as GPUs and more. You can get a list of configurations by typing this command into the terminal: bsub -h
        - The output of a job will not be displayed after its completion/termination. You have to specify an output file to save the result of the submitted job to. In this file, a description of the job and the possible script outputs/errors are provided. The syntax to specify this output file is given below

        - **bsub -o PATH_TO_OUTPUT_FILE.txt APPLICATION_NAME.EXTENSION**
            - Runs the APPLICATION_NAME.extension script and saves its output to the PATH_TO_OUTPUT_FILE.txt file.
            - Example: to run the python script file named my_script.py and save its output to the file named output_example.txt we can run the line below
                - bsub -o /gpfs/home/mblundon/output_example.txt python3 my_script.py

        - **bsub -J "JOB_NAME" APPLICATION_NAME.EXTENSION**
            - Runs the APPLICATION_NAME.EXTENSION script and names the job as JOB_NAME
            - Example: to run the python script file named my_script.py and name its job "first_job" we run the line below
                - bsub -J "first_job" python3 my_script.py

        - **bsub -q "QUEUE_NAME" APPLICATION_NAME.EXTENSION**
            - Submits the APPLICATION_NAME.extension script to the specified queue named QUEUE_NAME

        - Example: to submit the python script file named my_script.py to the large_memory queue we run the line below
            - bsub -q "large_memory" python3 my_script.py

- In order assign a GPU job you need to configure the GPU resources. An explanation of how to submit a basic job using GPUs is given below.

1. **bsub -q "NAME_OF_THE_GPU_QUEUE" -gpu - APPLICATION_NAME.EXTENSION**
    - Submits the "APPLICATION_NAME.EXTENSION" script to the queue named "NAME_OF_THE_GPU_QUEUE" and uses the GPU resources
    - Example: to submit the python script file named my_script.py to the GPU queue while utilizing the GPU resources we run the command below
        - bsub -q gpu -gpu - python3 my_script.py

- Other GPU resource configurations are shown at the end of this document.
- Below an example is shown that uses multiple configurations simultaneously.
    - **bsub -J "exampleJobName" -q "gpu" -o /gpfs/home/mblundon/output_example.txt -gpu - python3 my_script.py**
    - This example runs the python script named my_script.py through submitting a job named "exampleJobName" to the queue named "gpu" and uses GPU resources. It also saves the outputs of the job to the output_example.txt file in the given directory.

- **bkill JOB_ID**
    - Kills the job with the specified JOB_ID

- **cat OUTPUT_FILENAME.txt**
    - Reads the output file named OUTPUT_FILENAME.txt that was created through running a job

Other GPU resource configurations:

- bsub -gpu - | "[num=*num_gpus*[/task | host]] [:mode=shared | exclusive_process] [:mps=yes[,shared][,nocvd] | no | per_socket[,shared][,nocvd] | per_gpu[,shared][,nocvd]] [:j_exclusive=yes | no] [:aff=yes | no] [:block=yes | no] [:gpack=yes | no] [:glink=yes] [:gvendor=amd | nvidia] [:gmodel=*model_name*[-*mem_size*]] [:gtile=*tile_num*|'!'] [:gmem=*mem_value*]"

Documentation of every GPU parameter

- **num=*num_gpus*[/task | host]**
    - The number of physical GPUs required by the job. By default, the number is per host. You can also specify that the number is per task by specifying /task after the number.
- **mode=shared | exclusive_process**
    - The GPU mode when the job is running, either *shared* or *exclusive_process* . The default mode is *shared*. The *shared* mode corresponds to the Nvidia or AMD *DEFAULT* compute mode. The *exclusive_process* mode corresponds to the Nvidia *EXCLUSIVE_PROCESS* compute mode
- **mps=yes[,nocvd][,shared] | per_socket[,shared][,nocvd] | per_gpu[,shared][,nocvd] | no**

- Enables or disables the Nvidia Multi-Process Service (MPS) for the GPUs that are allocated to the job.
- If mps=yes is set, LSF starts one MPS daemon per host per job. When share is enabled (that is, if mps=yes,shared is set), LSF starts one MPS daemon per host for all jobs that are submitted by the same user with the same resource requirements. These jobs all use the same MPS daemon on the host.
- If mps=per_socket is set, LSF starts one MPS daemon per socket per job. When enabled with share (that is, if mps=per_socket,shared is set), LSF starts one MPS daemon per socket for all jobs that are submitted by the same user with the same resource requirements. These jobs all use the same MPS daemon for the socket.
- If mps=per_gpu is set, LSF starts one MPS daemon per GPU per job. When enabled with share (that is, if mps=per_gpu,shared is set), LSF starts one MPS daemon per GPU for all jobs that are submitted by the same user with the same resource requirements. These jobs all use the same MPS daemon for the GPU.

- **j_exclusive=yes | no**
  - Specifies whether the allocated GPUs can be used by other jobs. When the mode is set to *exclusive_process*, the *j_exclusive=yes* option is set automatically.
- **aff=yes | no**
  - Specifies whether to enforce strict GPU-CPU affinity binding. If set to no, LSF relaxes GPU affinity while maintaining CPU affinity. By default, aff=yes is set to maintain strict GPU-CPU affinity binding.
- **block=yes | no**
  - Specifies whether to enable block distribution, that is, to distribute the allocated GPUs of a job as blocks when the number of tasks is greater than the requested number of GPUs. If set to yes, LSF distributes all the allocated GPUs of a job as blocks when the number of tasks is bigger than the requested number of GPUs. By default, block=no is set so that allocated GPUs are not distributed as blocks.
- **gmodel=model_name**
  - Requests GPUs with the specified brand and model name. Here you can details about model using the commands given in GPU Resources Configuration.
- **gmem=mem_value**
  - Specify the GPU memory on each GPU required by the job. You can figure out the memory using the command **lsload - gpuload** which is provided in GPU Resources Configuration.