

llama.cpp RPC Multi-GPU Setup Guide (Windows 11 & Linux Distributed Cluster)

This was generated by NOTEBOOKLM from a lot of sources.

Question

I have three computers at home. one AMX Ryzen 7800 X3D with 32GB RAM, NVIDIA RTX 5070 12GB VRAM and windows 11. Two machines with an intel i5 11th gen, 16GB RAM and NVIDIA RTX 2070 tier 8GB VRAM.

Show or create a document which gives me extract instructions on how to setup llama rpc utilizing all three machines. which OS should I install on the two smaller intel i5 machines. how do i setup the llama rpc to utilize the total of 28GB VRAM for a suitable model.

Prompt used by NotenookLM

A technical guide with exact instructions on setting up llama.cpp RPC across one Windows 11 host and two Linux worker nodes, pooling 28GB of VRAM.

Answer

This technical deployment guide details the configuration of a high-performance distributed inference cluster. We will shard a Large Language Model (LLM) across a heterogeneous environment comprising one Windows 11 Master node and two Linux Worker nodes, pooling a total of 28GB of VRAM.

System Architecture & VRAM Pooling Strategy

In a distributed RPC (Remote Procedure Call) architecture, the model is sharded across the network. The Windows 11 Master node acts as the orchestrator; it physically hosts the .gguf model file and manages the Key-Value (KV) cache. The Linux Worker nodes act as "virtual GPUs," receiving tensor calculation tasks over the network, executing them on local hardware, and returning the results.

The Memory Math: 70B Parameter Constraints

Running a 70B model on a 28GB VRAM pool requires an aggressive offloading strategy. A standard 70B model quantized at Q4_K_M typically requires 42–43 GB just for the weights.

Component	Memory Requirement (70B Q4_K_M)	Pooled VRAM (28GB)	VRAM Deficit (System RAM Usage)
Model Weights	~43 GB	28 GB	-15 GB
KV Cache (8K Context)	~5 GB	0 GB	-5 GB
KV Cache (32K Context)	~25 GB	0 GB	-25 GB
Total Deficit	48–68 GB	--	-20 to -40 GB

Architect's Note: Because the model weights (43GB) exceed your physical VRAM pool (28GB), approximately 15GB of weights—plus the entirety of the KV cache—will reside in the Master node's system RAM. Expect a performance ceiling of 8–12 tokens per second (TPS) for a single stream, heavily dependent on the Master node's CPU and memory bandwidth.

The Memory Bandwidth Lever

During the decode phase (token generation), performance is strictly memory-bound. While the prefill phase (prompt processing) is compute-heavy, generation speed scales with how fast weights can be streamed from memory. In this distributed setup, your primary performance bottleneck is the movement of data between the system RAM, the local GPU, and the remote RPC workers.

Environment Preparation & Building llama.cpp

See also: <https://linuxconfig.org/debian-13-nvidia-driver-installation>

Linux Worker Nodes (Ubuntu/Debian)

Ensure both worker nodes are updated and have the NVIDIA stack correctly initialized.

1. Toolchain: `sudo apt install build-essential g++ cmake git curl libcurl4-openssl-dev pciutils`
2. Install NVIDIA Drivers

- Update APT Sources List with non-free

```
sudo sed -i 's/main/main non-free contrib/g' /etc/apt/sources.list
```

- sudo apt update

- Next install the required header

```
sudo apt install linux-headers-$(uname -r) build-essential dkms nvidia-detect
```

- run nvidia-detect to check if GPU is detected

- If card detected, install drivers

```
sudo apt install nvidia-driver nvidia-kernel-dkms
```

- Select OK when asked about the "wrong" kernel.

- sudo reboot

- After reboot run nvidia-smi to verify

3. CUDA Toolkit: Install nvidia-cuda-toolkit (required for the NVCC compiler).

```
sudo apt install nvidia-cuda-toolkit
```

4. Compute Capability: Identify your GPU's architecture:

```
nvidia-smi --query-gpu=compute_cap --format=csv
```

Windows 11 Master Node

The Master node must run a version of llama-cli.exe that matches the build version of the Linux workers exactly.

- Version Sync: Ensure all nodes are built from the same Git commit to avoid RPC protocol mismatches.
- Build Method: It is recommended to use the pre-compiled Windows releases with CUDA support or build via CMake to ensure llama-cli.exe is optimized for your local GPU.

Download and build LLAMA.cpp

<https://github.com/ggml-org/llama.cpp>

```
git clone https://github.com/ggml-org/llama.cpp
cd llama.cpp

# following the CUDO build here https://github.com/ggml-
org/llama.cpp/blob/master/docs/build.md#cuda
cmake -B build -DGGML_CUDA=ON
```

```
carsten@llama-w01:~/llama.cpp$ cmake -B build -DGGML_CUDA=ON
-- The C compiler identification is GNU 14.2.0
-- The CXX compiler identification is GNU 14.2.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
CMAKE_BUILD_TYPE=Release
-- Found Git: /usr/bin/git (found version "2.47.3")
-- The ASM compiler identification is GNU
-- Found assembler: /usr/bin/cc
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD - Success
-- Found Threads: TRUE
-- Warning: ccache not found - consider installing it for faster compilation or disable this warni
ng with GGML_CCACHE=OFF
-- CMAKE_SYSTEM_PROCESSOR: x86_64
-- GGML_SYSTEM_ARCH: x86
-- Including CPU backend
-- Found OpenMP_C: -fopenmp (found version "4.5")
-- Found OpenMP_CXX: -fopenmp (found version "4.5")
-- Found OpenMP: TRUE (found version "4.5")
-- x86 detected
-- Adding CPU backend variant ggml-cpu: -march=native
-- Found CUDAToolkit: /usr/include (found version "12.4.131")
-- CUDA Toolkit found
-- The CUDA compiler identification is NVIDIA 12.4.131 with host compiler GNU 13.3.0
-- Detecting CUDA compiler ABI info
-- Detecting CUDA compiler ABI info - done
-- Check for working CUDA compiler: /usr/bin/nvcc - skipped
-- Detecting CUDA compile features
-- Detecting CUDA compile features - done
-- Using CMAKE_CUDA_ARCHITECTURES=75-real CMAKE_CUDA_ARCHITECTURES_NATIVE=75-real
-- CUDA host compiler is GNU 13.3.0
-- Including CUDA backend
-- ggml version: 0.9.11
-- ggml commit: d12cc3d1c
-- Found OpenSSL: /usr/lib/x86_64-linux-gnu/libcrypto.so (found version "3.5.5")
-- Performing Test OPENSSL_VERSION_SUPPORTED
-- Performing Test OPENSSL_VERSION_SUPPORTED - Success
-- OpenSSL found: 3.5.5
-- Generating embedded license file for target: common
-- Configuring done (6.6s)
-- Generating done (0.2s)
-- Build files have been written to: /home/carsten/llama.cpp/build
carsten@llama-w01:~/llama.cpp$ █

carsten@llama-w02:~/llama.cpp$ cmake -B build -DGGML_CUDA=ON
-- The C compiler identification is GNU 14.2.0
-- The CXX compiler identification is GNU 14.2.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
CMAKE_BUILD_TYPE=Release
-- Found Git: /usr/bin/git (found version "2.47.3")
-- The ASM compiler identification is GNU
-- Found assembler: /usr/bin/cc
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD - Success
-- Found Threads: TRUE
-- Warning: ccache not found - consider installing it for faster compilation or disable this warni
ng with GGML_CCACHE=OFF
-- CMAKE_SYSTEM_PROCESSOR: x86_64
-- GGML_SYSTEM_ARCH: x86
-- Including CPU backend
-- Found OpenMP_C: -fopenmp (found version "4.5")
-- Found OpenMP_CXX: -fopenmp (found version "4.5")
-- Found OpenMP: TRUE (found version "4.5")
-- x86 detected
-- Adding CPU backend variant ggml-cpu: -march=native
-- Found CUDAToolkit: /usr/include (found version "12.4.131")
-- CUDA Toolkit found
-- The CUDA compiler identification is NVIDIA 12.4.131 with host compiler GNU 13.3.0
-- Detecting CUDA compiler ABI info
-- Detecting CUDA compiler ABI info - done
-- Check for working CUDA compiler: /usr/bin/nvcc - skipped
-- Detecting CUDA compile features
-- Detecting CUDA compile features - done
-- Using CMAKE_CUDA_ARCHITECTURES=75-real CMAKE_CUDA_ARCHITECTURES_NATIVE=75-real
-- CUDA host compiler is GNU 13.3.0
-- Including CUDA backend
-- ggml version: 0.9.11
-- ggml commit: d12cc3d1c
-- Found OpenSSL: /usr/lib/x86_64-linux-gnu/libcrypto.so (found version "3.5.5")
-- Performing Test OPENSSL_VERSION_SUPPORTED
-- Performing Test OPENSSL_VERSION_SUPPORTED - Success
-- OpenSSL found: 3.5.5
-- Generating embedded license file for target: common
-- Configuring done (6.5s)
-- Generating done (0.2s)
-- Build files have been written to: /home/carsten/llama.cpp/build
carsten@llama-w02:~/llama.cpp$ █
```

Next build the tool

```
# Use j parameter to have multiple jobs parallel
# It will speed things up significantly.
# Put the number of your available cores in
cmake --build build --config Release -j 8
```



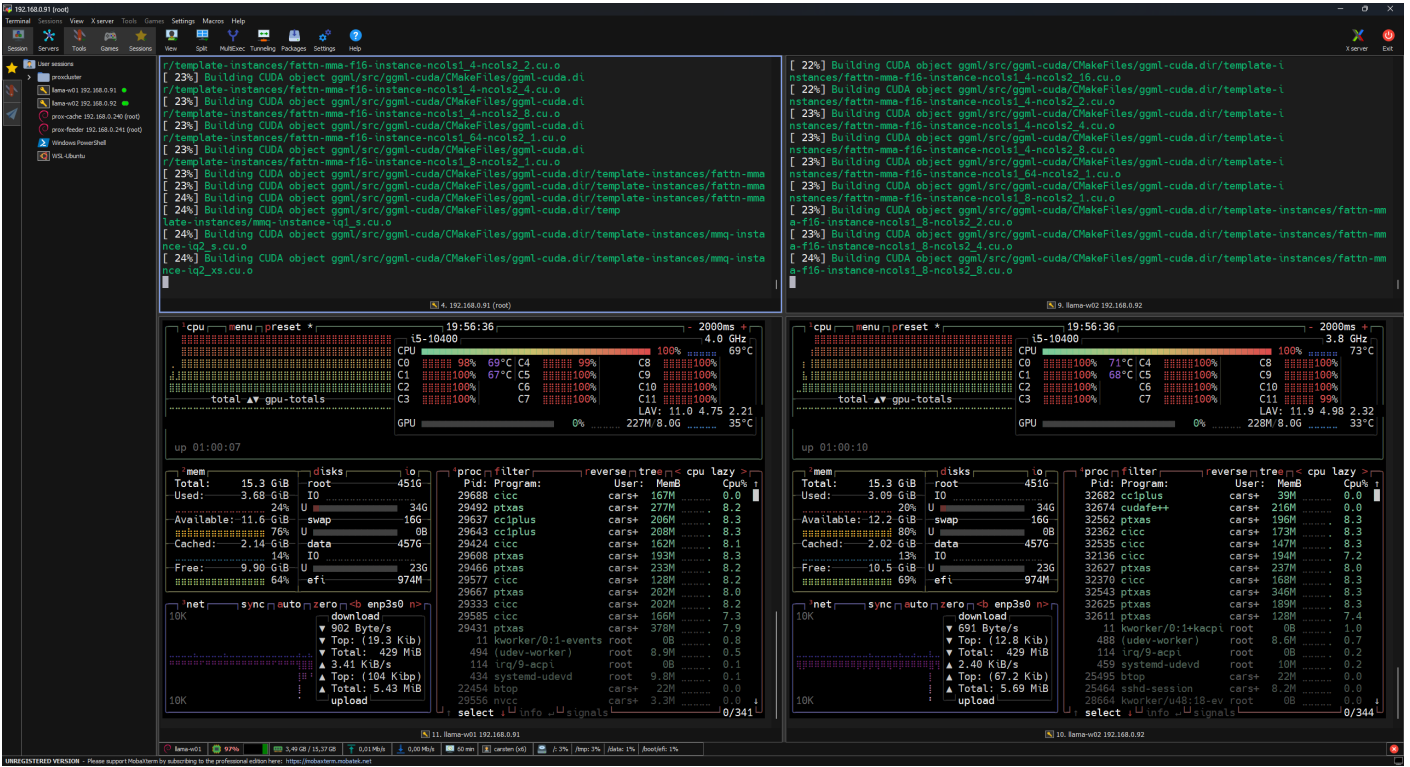
Running `bttop` in another terminal to see if something is still happening.

Compiling again with the `RPC=ON` flag following these instructions: <https://github.com/ggml-org/llama.cpp/tree/master/tools/rpc#remote-hosts>

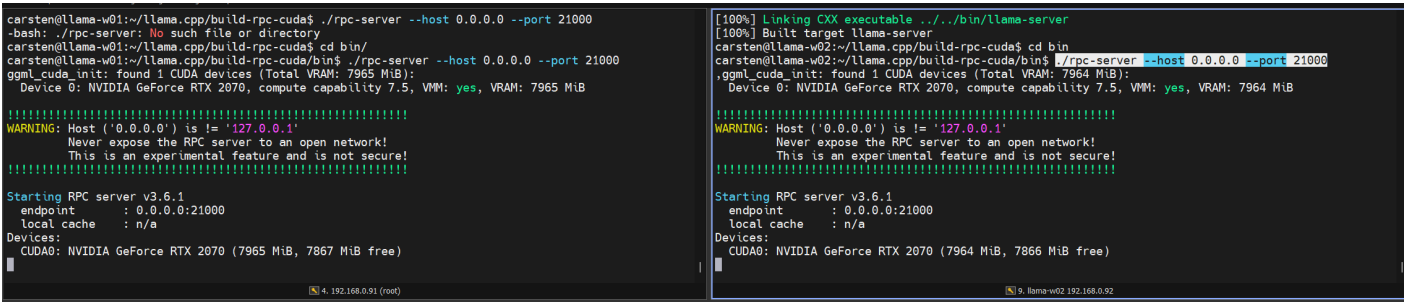
In the `llama.cpp` directory (the git clone) run the following cmds

```
mkdir build-rpc-cuda
cd build-rpc-cuda
cmake .. -DGGML_CUDA=ON -DGGML_RPC=ON
cmake --build . --config Release
```

This will build the `rpc-server` executable that we need.



After the build finished, we can change into the bin folder and run the rpc-server



Download and build LLAMA.cpp on Windows

Install with WINGET

Networking & Security Requirements

Do not use Wi-Fi. Distributed inference is highly sensitive to network latency. Wi-Fi introduces jitter and high latency that will degrade token generation to unusable speeds. Wired Gigabit Ethernet is the mandatory baseline for a stable RPC cluster.

Security Protocols

The llama.cpp RPC protocol transmits unencrypted model tensors and hidden states. Exposing these ports to the WAN is a critical security vulnerability.

- Internal Only: Restrict RPC traffic to trusted internal IPs or a secured VPC.
- Firewall: Ensure the chosen ports (default 50052 or custom) are open on Linux workers for the Master's IP.

Linux Worker Configuration (RPC Server)

Each worker creates a socket to listen for incoming data (model parameters and hidden states) from the master.

Command to initialize:

```
./rpc-server --host 0.0.0.0 --port [PORT]
```

The --host 0.0.0.0 flag allows the server to listen on all interfaces, which is safe only if your local firewall/security groups are configured to restrict access.

5. Windows 11 Master Node Configuration

The Master node orchestrates the sharding by treating remote nodes as additional CUDA devices.

Launching Distributed Inference:

```
./llama-cli.exe -m llama-3-70b-q4_k_m.gguf --rpc  
[WORKER_1_IP]:[PORT],[WORKER_2_IP]:[PORT] --n-gpu-layers 100 --ctx-size 8192
```

- --rpc: Points to the remote "virtual GPUs."

- `--n-gpu-layers 100`: High value ensures the engine attempts to offload every layer possible into the pooled VRAM before falling back to system RAM.
-

6. Performance Tuning & Optimization

To maximize tokens per second in an offloading scenario, use the following "Quick Win" flags:

- `--flash-attn`: Enables Flash Attention to reduce the memory footprint and compute time of the attention mechanism.
- `--mlock`: Force-locks the model in RAM, preventing the operating system from swapping weights to the disk (pagefile), which is a common cause of stuttering.
- `--no-mmap`: Recommended for the worker nodes if they have fast NVMe drives but limited system RAM; it forces a full load of the shard into memory at startup.
- `--threads`: Match the thread count to the physical CPU cores for the Master's fallback processing.

For GPU-assisted or distributed inference, the optimal thread count is typically 4–8 threads, regardless of the total core count. Over-allocating threads often leads to context-switching overhead that reduces performance.

7. Troubleshooting & Connectivity Diagnostics

Error: "RPC Failed to Connect" or "0 MiB free"

If the Master connects but the worker reports "0 MiB free" on the device:

1. **Backend Failure**: This usually indicates the RPC server is reachable, but the CUDA backend failed to initialize the GPU.
2. **Arch Mismatch**: Check if `CUDA_DOCKER_ARCH` was set correctly during the build on the Linux worker.
3. **Toolkit Issue**: Ensure the `nvidia-cuda-toolkit` is properly installed and `nvidia-smi` shows no errors.

Version Mismatch

If the nodes connect but crash during tensor loading, verify that the llama.cpp version (Git hash) is identical on Windows and Linux. RPC is under active development and protocol changes are frequent.

8. Advancing the Setup: RAG Integration

Because the 28GB pool is insufficient for the weights and KV cache of a 70B model, Retrieval-Augmented Generation (RAG) is the superior strategy for handling large datasets.

Instead of extending the context window (which exponentially consumes VRAM and increases latency), RAG pulls relevant text chunks from an external database only when needed. This keeps your context window small (e.g., 8K), preserves VRAM for the model weights, and prevents "Out of Memory" (OOM) errors while maintaining high accuracy for documentation-heavy tasks.

Some Screenshots

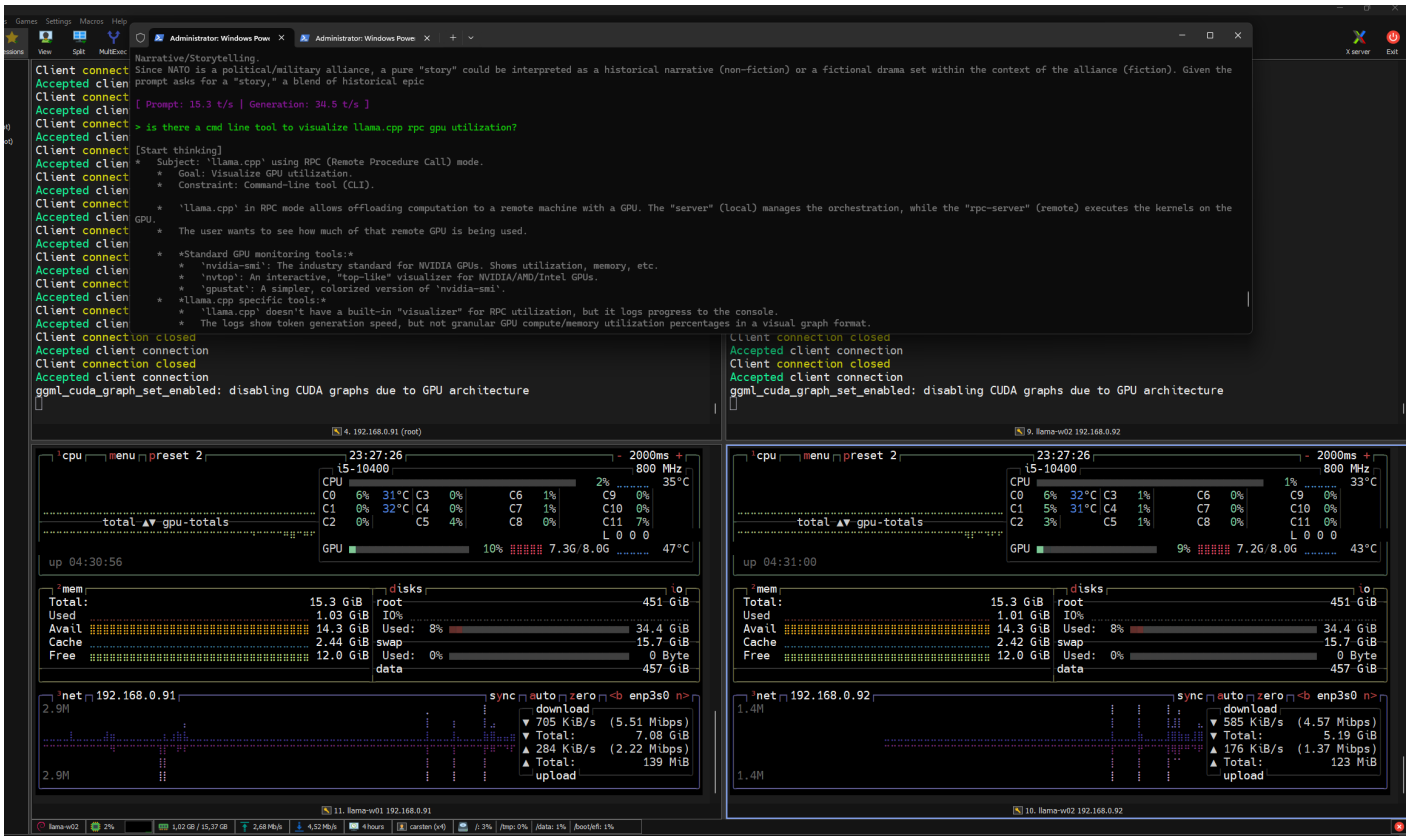
```
Linux llama-w01 6.12.74+deb13+1-amd64 #1 SMP PREEMPT_DYNAMIC Debian 6.12.74-2 (2026-03-08)
carsten@llama-w01:~$ nvidia-smi
NVIDIA-SMI 550.163.01 Driver Version: 550.163.01 CUDA Version: 12.4
+-----+-----+-----+-----+-----+-----+
| GPU   | Name      | Persistence-M | Bus-Id  | Disp.A | Volatile Uncorr. ECC |
| Fan  | Temp     | Perf         | Pwr:Usage/Cap | Memory-Usage | GPU-Util  | Compute M. |
|-----+-----+-----+-----+-----+-----+
| 0     | NVIDIA GeForce RTX 2070 | Off         | 00000000:01:00:0 | Off      | 0%        | Default     |
| 34%  | 42C     | P8          | 17W / 175W    | 7246MiB / 8192MiB |           | N/A        |
+-----+-----+-----+-----+-----+-----+
Processes:
GPU   GI   CI   PID  Type  Process name      GPU Memory
ID   ID   ID                   Usage
-----+-----+-----+-----+-----+
0     N/A  N/A  39166  C    ./rpc-server      7242MiB
carsten@llama-w01:~$
```

```
llama.cpp
build      : b8693-d8a6df62
model     : ggml-org/gemma-4-26B-A4B-it-GGUF:Q4_K_M
modalities: text, vision

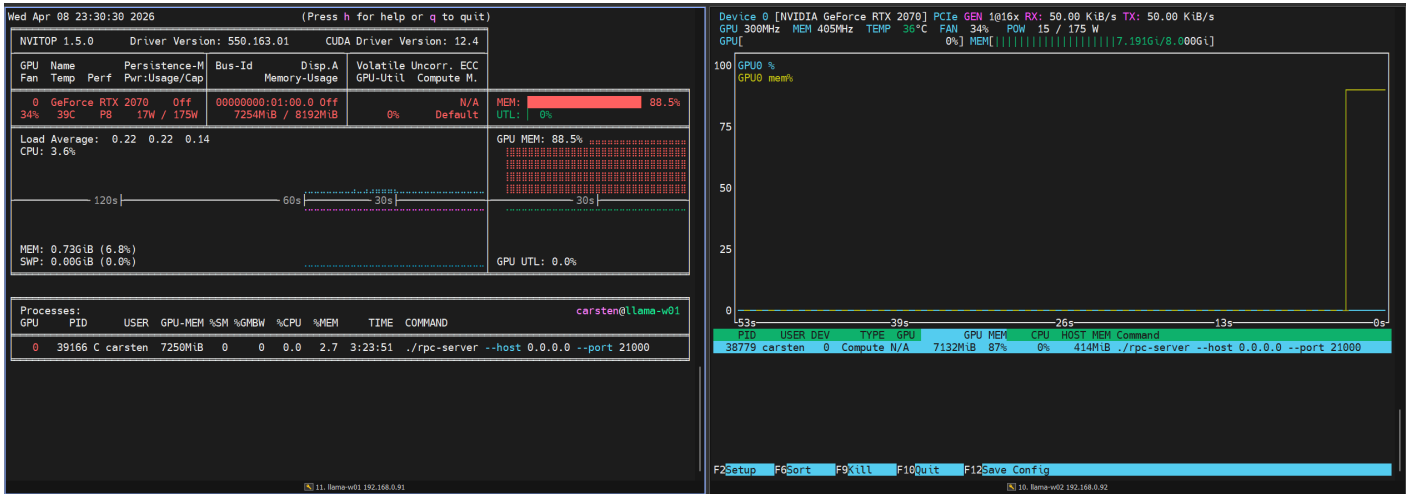
available commands:
./exit or Ctrl+C      stop or exit
./regen              regenerate the last response
./clear              clear the chat history
./read <file>        add a text file
./glob <pattern>     add text files using globbing pattern
./image <file>       add an image file

> Hello. Who are you?
[Start thinking]
* User asks: "Hello. Who are you?"
* Goal: Identify myself according to the system instructions.
* Name: Gemma 4.
* Developer: Google DeepMind.
* Nature: Large Language Model.
* Type: Open weights model.
* Capabilities: Text and image processing (audio for 2B/4B), text output only.
* Knowledge cutoff: January 2025.
* "I am Gemma 4, a large language model developed by Google DeepMind."
* The system instructions state: "When asked about your identity, always refer to yourself as Gemma 4."
* Include developer and nature.
>Hello! I am Gemma 4, a large language model developed by Google DeepMind."
[End thinking]
Hello! I am Gemma 4, a large language model developed by Google DeepMind.

[ Prompt: 10.0 t/s | Generation: 34.5 t/s ]
> Can you write python code?
[Start thinking]
The user is asking if I can write Python code.
* I am Gemma 4, a large language model.
* I can process text and images.
* I can generate text (which includes code).
* I am capable of coding tasks as a language model.
Yes, I can write Python code. I should offer to help with a specific task or provide a simple example to demonstrate.
*Drafting response:*
"Yes, I can write Python code. Whether you need help with a simple script, data analysis, web scraping, or solving complex algorithms, I can assist you.
Do you have a specific task or problem in mind that you'd like me to write some code for?"
[End thinking]
Yes, I can write Python code. I can help you with a wide range of tasks, including:
* **Writing scripts** for automation.
* **Data analysis and manipulation** (using libraries like Pandas or NumPy).
* **Implementing algorithms** and solving coding challenges.
```



NVTOP



NVITOP

nvitop 1.5.0 - (C) Xuehai Pan, 2021-2025.
Released under the GNU GPLv3 License.

GPU Process Type: C: Compute, G: Graphics, X: Mixed.

Device coloring rules by loading intensity:

- GPU utilization: light < 10% <= moderate < 75% <= heavy.
- GPU-MEM percent: light < 10% <= moderate < 80% <= heavy.

a f c: change display mode h ?: show this help screen
F5 r: force refresh window q: quit

Arrows: scroll process list Space: tag/untag current process
Home: select the first process Esc: clear process selection
End: select the last process Ctrl-C I: interrupt selected process

Ctrl-A ^: scroll to left most K: kill selected process
Ctrl-E \$: scroll to right most T: terminate selected process
PageUp [: scroll entire screen up e: show process environment
PageDown]: scroll entire screen down t: toggle tree-view screen
Enter: show process metrics

Wheel: scroll process list Shift-Wheel: scroll horizontally
Tab: scroll process list Ctrl-Wheel: fast scroll (5x)

on oN: sort by GPU-INDEX ob oB: sort by %GMBW
op oP: sort by PID oc oC: sort by %CPU
ou oU: sort by USER om oM: sort by %MEM
og oG: sort by GPU-MEM ot oT: sort by TIME
os oS: sort by %SM /.: invert sort order
, .: select sort column

Press any key to return.

11. llama-w01 192.168.0.91

NVITOP, NVTOP and Terminal with llama-cli

The screenshot shows a Windows Task Manager window on the left, displaying system performance for a NVIDIA GeForce RTX 5070. The GPU usage is at 1%, with 10.1/12.0 GB of memory used. The system is running on a Windows 11 Home (x64) edition.

On the right, a terminal window shows the installation and usage of three tools for GPU monitoring:

- ## 1. 'nvtop' (The Most Visual)**: A command-line tool that provides bar graphs for GPU utilization, memory usage, and a list of processes. Installation: `sudo apt install nvtop` (Ubuntu/Debian) or `brew install nvtop` (macOS). Usage: `nvtop`.
- ## 2. 'nvtop' (The "Pro" Choice)**: A more modern, feature-rich version of 'nvtop' written in Python. Installation: `pip install nvtop`. Usage: `nvtop`.
- ## 3. 'nvidia-smi' with 'watch' (The "No-Install" Choice)**: A static snapshot of GPU usage. Usage: `watch nvidia-smi`.

Below the terminal, two instances of the 'nvtop' tool are shown. The first instance shows a GPU memory usage of 88.5% and a CPU usage of 0.8%. The second instance shows a GPU memory usage of 87.1% and a CPU usage of 1.2%. Both instances show a list of processes running on the GPU, with the first process being `./rpc-server`.

After running

```

Exiting...
llama_memory_breakdown_print: | memory breakdown [MiB] | total free self model context compute unaccounted |
llama_memory_breakdown_print: | - RPC0 (192.168.0.91:21000) | 7965 = 705 + (7120 = 5168 + 1132 + 819) + 139 |
llama_memory_breakdown_print: | - RPC1 (192.168.0.92:21000) | 7964 = 822 + (7002 = 4517 + 2132 + 353) + 138 |
llama_memory_breakdown_print: | - Vulkan0 (RTX 5070) | 11855 = 2 + (8994 = 6316 + 2156 + 521) + 2858 |
llama_memory_breakdown_print: | - Host | 1103 = 577 + 0 + 526 |
PS C:\Users\Admin> |
model: Load 18050951: ATTACHED 31/31 LAYERS TO GPU
  
```

Revision #8

Created 2026-04-08 12:36:07 UTC by Carsten

Updated 2026-04-08 21:36:26 UTC by Carsten