Recently, I wanted to adjust the frequency of my NVIDIA GPU on Linux because I noticed some software (like FaceFusion) doesn't utilize the GPU at its maximum frequency (e.g., a maximum of 2000 MHz, but only 1850 MHz in practice). So I decided to figure out how to tweak settings like frequency. This article will use `nvtop` and `nvidia-smi` for monitoring.

I'll first explain basic frequency adjustment, then cover undervolting and overclocking, and finally share adjustment strategies and precautions.

> Linux has a policy table that the GPU follows for dynamic adjustments unless you manually lock the frequency. All settings in this article comply with this behavior.
> After adjusting frequencies as described in this section, you may notice both core and memory frequencies seem constrained by certain factors. For example, I set my memory frequency to `7500` MHz (it stays at 7500 MHz when idle), but drops to `7300` MHz when running some programs. If you know why this happens, please let me know—or feel free to share your hypotheses for discussion.

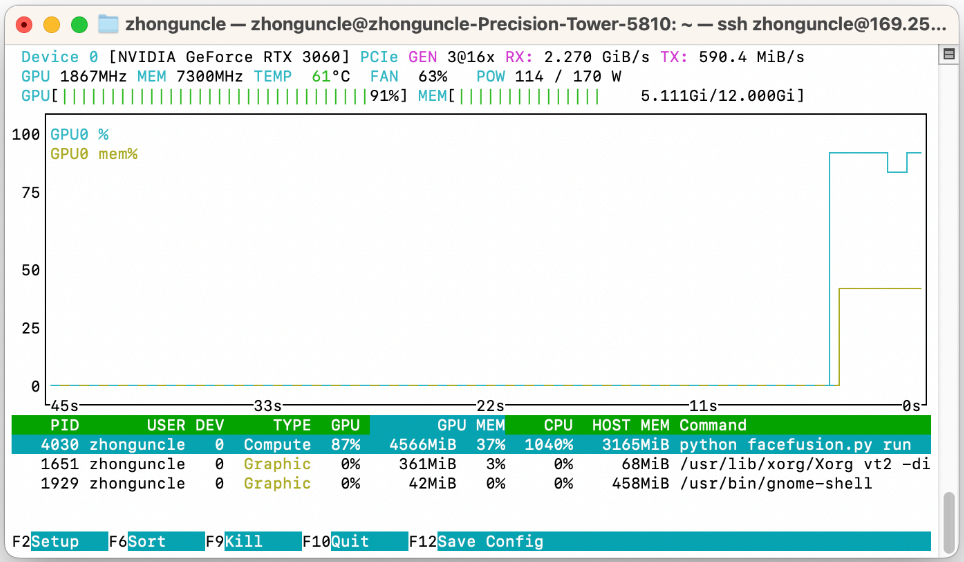
Additionally, modifications on Linux reset after reboot, so you'll need to reapply them each time. You can add them to your `cron` jobs or startup scripts (I won't cover this here to keep the article concise).

Adjusting Frequencies, Power Limits, etc. (via `nvidia-smi`)

First, you need to get the GPU's UUID or PCI Bus ID to specify which card to modify.

> If you only have one GPU, you can simply use `0`.

Let's first check the frequency with `nvtop` (while running FaceFusion):



zhonguncle — zhonguncle@zhonguncle-Precision-Tower-5810: ~ — ssh zhonguncle@169.25...

```
Device 0 [NVIDIA GeForce RTX 3060] PCIe GEN 3@16x RX: 2.270 GiB/s TX: 590.4 MiB/s
GPU 1867MHz MEM 7300MHz TEMP  61°C  FAN  63%   POW 114 / 170 W
GPU[||||||||||||||||||||||||||||||91%] MEM[||||||||||||||     5.111Gi/12.000Gi]

    PID      USER DEV     TYPE  GPU        GPU  MEM    CPU  HOST MEM Command
   4030 zhonguncle   0  Compute  87%   4566MiB  37%  1040%   3165MiB python facefusion.py run
   1651 zhonguncle   0  Graphic   0%    361MiB   3%     0%     68MiB /usr/lib/xorg/Xorg vt2 -di
   1929 zhonguncle   0  Graphic   0%     42MiB   0%     0%    458MiB /usr/bin/gnome-shell

F2Setup  F6Sort  F9Kill  F10Quit  F12Save Config
```

You can see the frequency is `1867 MHz` and power consumption is `114W`.

List your GPUs with `nvidia-smi -L`—here the RTX 3060's UUID is `GPU-037a6092-3f62-5b68-ac66-856918216320`:

```
$ nvidia-smi -L
GPU 0: NVIDIA GeForce RTX 3060 (UUID: GPU-037a6092-3f62-5b68-ac66-856918216320)
```

Next, use `sudo nvidia-smi --id=[UUID] -lgc [min_freq],[max_freq]`. To lock the core frequency at 2000 MHz, set both min and max to `2000`:

> `-lgc` = lock core clock, `-lmc` = lock memory clock, `-pl` = power limit.

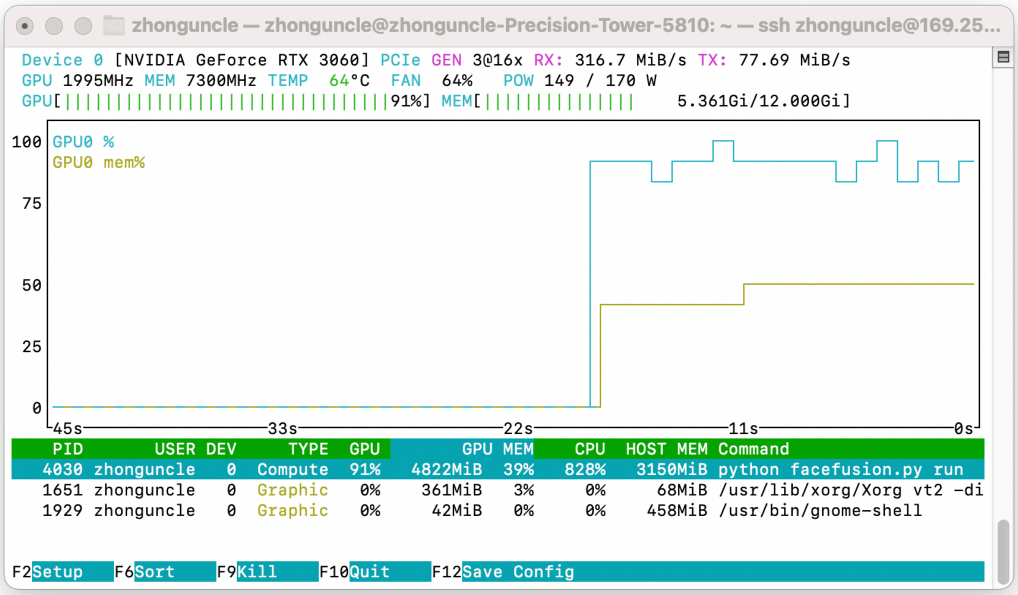
```
$ sudo nvidia-smi --id=GPU-037a6092-3f62-5b68-ac66-856918216320 -lgc 2000,2000
[sudo] password for zhonguncle:
GPU clocks set to "(gpuClkMin 2000, gpuClkMax 2000)" for GPU 00000000:03:00.0

Warning: persistence mode is disabled on device 00000000:03:00.0. See the Known Issues
section of the nvidia-smi(1) man page for more information. Run with [--help | -h] switch
to get more information on how to enable persistence mode.
All done.
```

> The `Warning` indicates persistence mode is disabled, meaning some tasks may not apply your changes.

To enable persistence mode, use `-pm 1`:

```
$ sudo nvidia-smi -pm 1
Enabled persistence mode for GPU 00000000:03:00.0.
All done.
```

Check the status again:



```
zhonguncle — zhonguncle@zhonguncle-Precision-Tower-5810: ~ — ssh zhonguncle@169.25...
Device 0 [NVIDIA GeForce RTX 3060] PCIe GEN 3@16x RX: 316.7 MiB/s TX: 77.69 MiB/s
GPU 1995MHz MEM 7300MHz TEMP  64°C  FAN  64%   POW 149 / 170 W
GPU[|||||||||||||||||||||||||||||||91%] MEM[||||||||||||||     5.361Gi/12.000Gi]

   PID       USER DEV     TYPE  GPU        GPU MEM     CPU  HOST MEM Command
  4030 zhonguncle   0  Compute  91%   4822MiB  39%    828%   3150MiB python facefusion.py run
  1651 zhonguncle   0  Graphic   0%    361MiB   3%      0%     68MiB /usr/lib/xorg/Xorg vt2 -di
  1929 zhonguncle   0  Graphic   0%     42MiB   0%      0%    458MiB /usr/bin/gnome-shell

F2Setup  F6Sort   F9Kill   F10Quit   F12Save Config
```

The frequency is now `1995 MHz` and power consumption is `149W`.

To modify the power limit, use `-pl`:

```
$ sudo nvidia-smi --id=GPU-037a6092-3f62-5b68-ac66-856918216320 -pl 120
Power limit for GPU 00000000:03:00.0 was set to 120.00 W from 170.00 W.
All done.
```

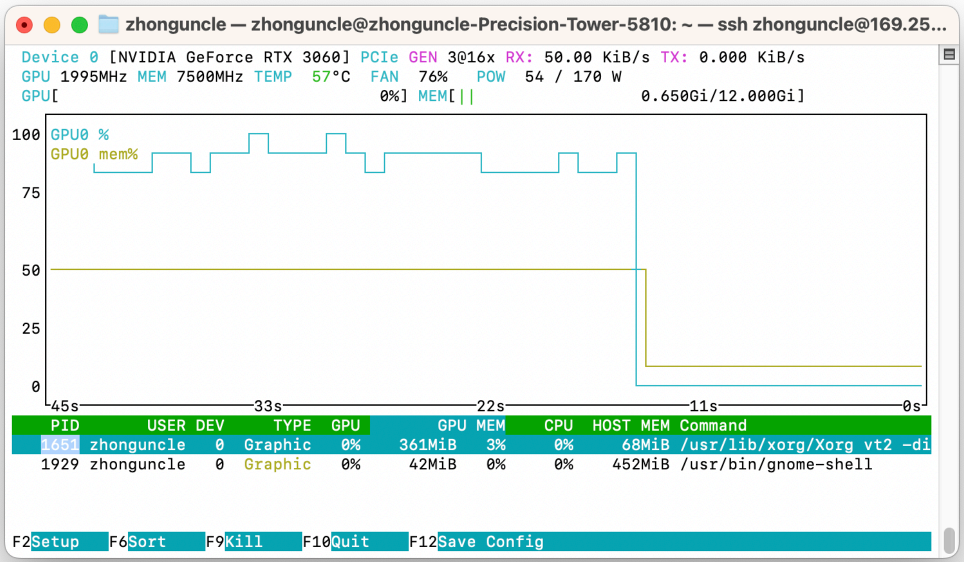
However, the frequency may become unstable at this point. If you set a value outside the valid range, you'll get a warning:

```
$ sudo nvidia-smi --id=GPU-037a6092-3f62-5b68-ac66-856918216320 -pl 200
Provided power limit 200.00 W is not a valid power limit which should be between 100.00 W
and 187.00 W for GPU 00000000:03:00.0
Terminating early due to previous errors.
```

> As shown, the valid power range is `between 100.00 W and 187.00 W`.

With this method, reducing power limit almost always lowers frequency. In the next section, we'll cover undervolting, which lets you reduce power consumption while maintaining or even improving performance.

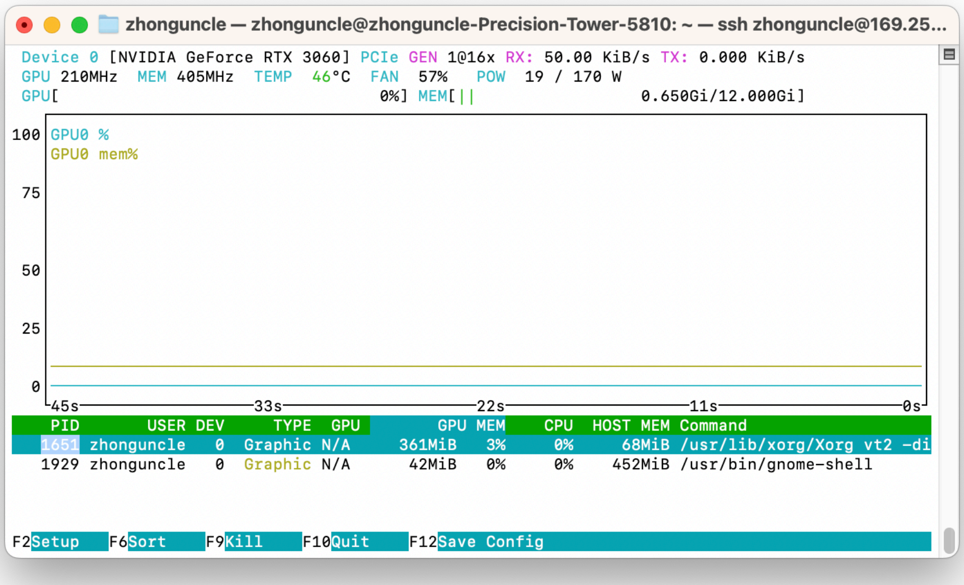
Additionally, with fixed frequency settings, power consumption remains high even at idle:


zhonguncle — zhonguncle@zhonguncle-Precision-Tower-5810: ~ — ssh zhonguncle@169.25...

```
Device 0 [NVIDIA GeForce RTX 3060] PCIe GEN 3@16x RX: 50.00 KiB/s TX: 0.000 KiB/s
GPU 1995MHz MEM 7500MHz TEMP  57°C  FAN  76%   POW  54 / 170 W
GPU[                                   0%] MEM[||                 0.650Gi/12.000Gi]

100 GPU0 %
    GPU0 mem%
 75

 50

 25

  0
   └45s─────────────────33s──────────────────22s──────────────────11s────────────────0s┘
   PID          USER DEV    TYPE  GPU        GPU  MEM     CPU  HOST MEM Command
   1651 zhonguncle   0  Graphic   0%     361MiB   3%      0%     68MiB /usr/lib/xorg/Xorg vt2 -di
   1929 zhonguncle   0  Graphic   0%      42MiB   0%      0%    452MiB /usr/bin/gnome-shell

F2Setup   F6Sort    F9Kill    F10Quit    F12Save Config
```

To reset frequencies, use `-rgc` (reset core clock) and `-rmc` (reset memory clock):

```
$ sudo nvidia-smi --id=GPU-037a6092-3f62-5b68-ac66-856918216320 -rgc
All done.
$ sudo nvidia-smi --id=GPU-037a6092-3f62-5b68-ac66-856918216320 -rmc
All done.
```


zhonguncle — zhonguncle@zhonguncle-Precision-Tower-5810: ~ — ssh zhonguncle@169.25...

```
Device 0 [NVIDIA GeForce RTX 3060] PCIe GEN 1@16x RX: 50.00 KiB/s TX: 0.000 KiB/s
GPU 210MHz  MEM 405MHz  TEMP  46°C  FAN  57%   POW  19 / 170 W
GPU[                                   0%] MEM[||                 0.650Gi/12.000Gi]

100 GPU0 %
    GPU0 mem%
 75

 50

 25

  0
   └45s─────────────────33s──────────────────22s──────────────────11s────────────────0s┘
   PID          USER DEV    TYPE  GPU        GPU  MEM     CPU  HOST MEM Command
   1651 zhonguncle   0  Graphic N/A      361MiB   3%      0%     68MiB /usr/lib/xorg/Xorg vt2 -di
   1929 zhonguncle   0  Graphic N/A       42MiB   0%      0%    452MiB /usr/bin/gnome-shell

F2Setup   F6Sort    F9Kill    F10Quit    F12Save Config
```

Like power limits, core and memory frequencies have predefined ranges. View the full list with:

```
nvidia-smi --query-supported-clocks=timestamp,gpu_name,gpu_uuid,memory,graphics --
format=csv
timestamp, gpu_name, gpu_uuid, memory [MHz], graphics [MHz]
2025/09/09 06:03:42.922, NVIDIA GeForce RTX 3060, GPU-037a6092-3f62-5b68-ac66-
856918216320, 7501 MHz, 2190 MHz
2025/09/09 06:03:42.922, NVIDIA GeForce RTX 3060, GPU-037a6092-3f62-5b68-ac66-
856918216320, 7501 MHz, 2182 MHz
2025/09/09 06:03:42.922, NVIDIA GeForce RTX 3060, GPU-037a6092-3f62-5b68-ac66-
856918216320, 7501 MHz, 2175 MHz
2025/09/09 06:03:42.922, NVIDIA GeForce RTX 3060, GPU-037a6092-3f62-5b68-ac66-
856918216320, 7501 MHz, 2167 MHz
....
```

This maximum core frequency is essentially the highest sustainable value (though this may be specific to my RTX 3060—feel free to test higher values).

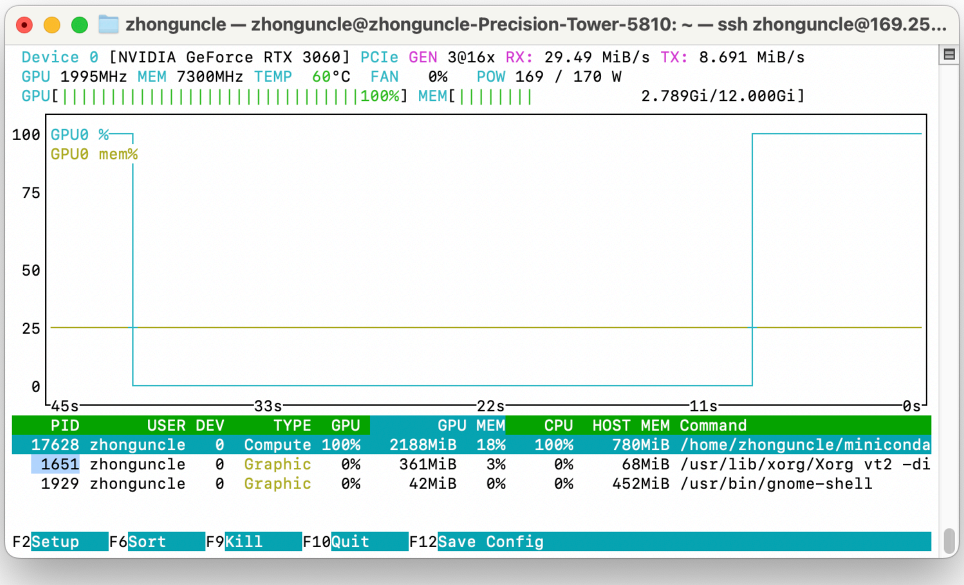
Undervolting and Overclocking (Python/pynvml)

Next, we'll cover undervolting and overclocking.

We'll use Python for this, as NVIDIA doesn't provide a command-line API for undervolting.

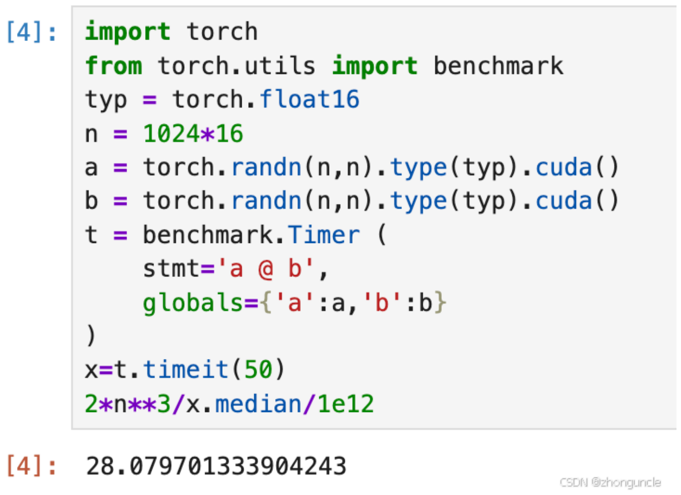
> Important note: We'll use root privileges, so proceed carefully and test incrementally. Avoid large adjustments to prevent irreversible damage.
> Since we're using NVML, you can also write a C/C++ version and compile it to run on startup.

First, the default state:



```
zhonguncle — zhonguncle@zhonguncle-Precision-Tower-5810: ~ — ssh zhonguncle@169.25...
Device 0 [NVIDIA GeForce RTX 3060] PCIe GEN 3@16x RX: 29.49 MiB/s TX: 8.691 MiB/s
GPU 1995MHz MEM 7300MHz TEMP  60°C  FAN   0%   POW 169 / 170 W
GPU[||||||||||||||||||||||||||||||100%] MEM[||||||||          2.789Gi/12.000Gi]

  PID       USER DEV     TYPE  GPU       GPU  MEM    CPU  HOST MEM Command
17628 zhonguncle   0  Compute 100%   2188MiB  18%   100%    780MiB /home/zhonguncle/miniconda
 1651 zhonguncle   0  Graphic   0%    361MiB   3%     0%     68MiB /usr/lib/xorg/Xorg vt2 -di
 1929 zhonguncle   0  Graphic   0%     42MiB   0%     0%    452MiB /usr/bin/gnome-shell

F2Setup   F6Sort    F9Kill    F10Quit    F12Save Config
```

FP16 benchmark results at this state: `28.08` TFLOPS:



```
[4]: import torch
     from torch.utils import benchmark
     typ = torch.float16
     n = 1024*16
     a = torch.randn(n,n).type(typ).cuda()
     b = torch.randn(n,n).type(typ).cuda()
     t = benchmark.Timer (
         stmt='a @ b',
         globals={'a':a,'b':b}
     )
     x=t.timeit(50)
     2*n**3/x.median/1e12

[4]: 28.079701333904243
```

Checking voltage with `nvidia-smi -q -d VOLTAGE` shows around 1050 mV.

Install `nvidia-ml-py` (using Python's pip—Python environments can be tricky!):

```
$ sudo python -m pip install nvidia-ml-py
Requirement already satisfied: nvidia-ml-py in /usr/local/lib/python3.10/dist-packages
(13.580.65)
WARNING: Running pip as the 'root' user can result in broken permissions and conflicting
behaviour with the system package manager. It is recommended to use a virtual environment
instead: https://pip.pypa.io/warnings/venv
```

Or install via apt:

```
sudo apt install python3-pynvml
```

Ignore the warning—we intentionally use root for this operation.

Create a script to dynamically adjust frequency (+100MHz), set a fixed power limit (-30W), and tweak voltage:

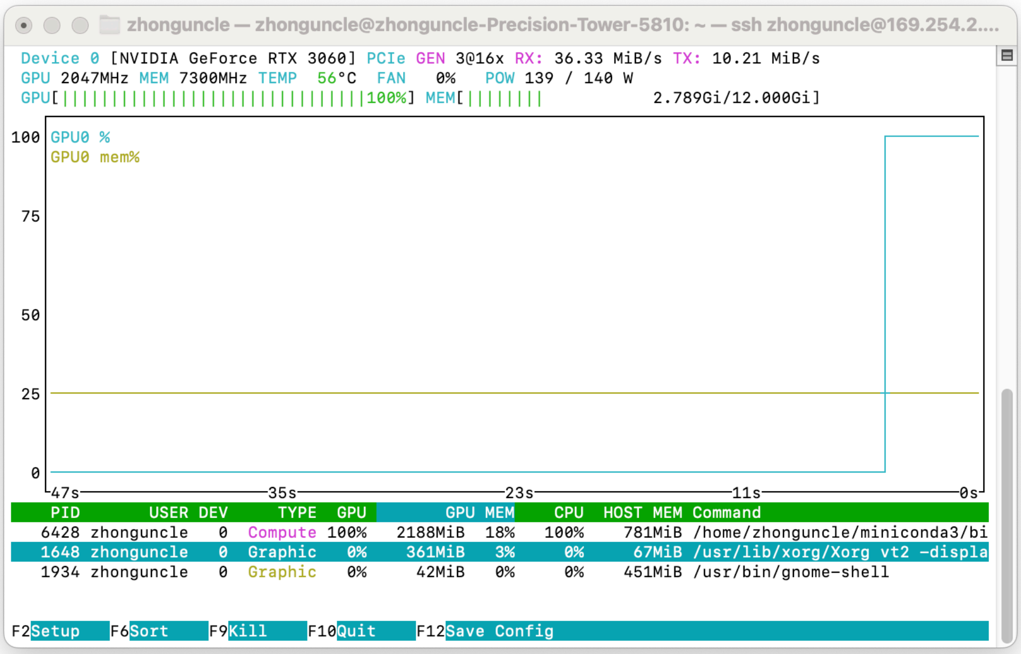
> I recommend reading the full section before running this script.

```
from pynvml import *
# Initialize NVML
nvmlInit()
GPU = nvmlDeviceGetHandleByIndex(0)

# Set core clock offset to +100 MHz
nvmlDeviceSetGpcClkVfOffset(GPU, 100)
# Set memory clock offset to 0 MHz
nvmlDeviceSetMemClkVfOffset(GPU, 0)
# Set power limit to 140 W (140000 mW)
nvmlDeviceSetPowerManagementLimit(GPU, 140000)
# Set GpcClk V/F (Voltage/Frequency) curve offset to -200 MHz.
# Unlike MSI Afterburner, we use this to indirectly reduce voltage (explained later).
nvmlDeviceSetGpcClkVfOffset(GPU, 200)

# Shutdown NVML
nvmlShutdown()
```

Run with `sudo` or as root:

```
$ sudo python3 oc.py
```

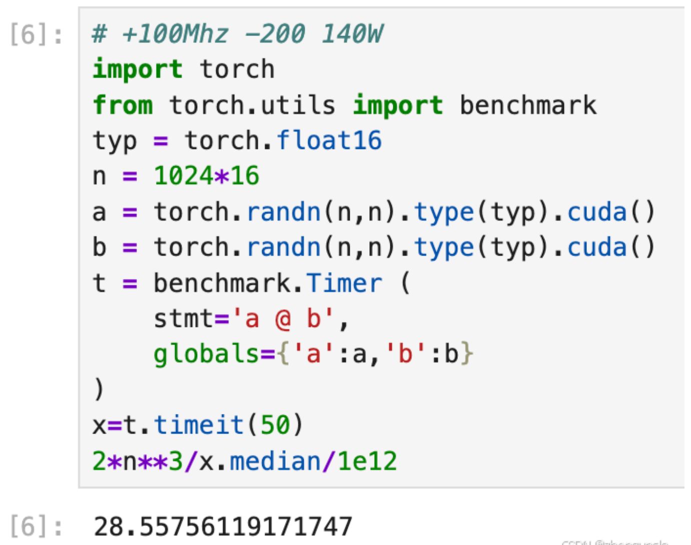
With everything configured, test the results:



```
Device 0 [NVIDIA GeForce RTX 3060] PCIe GEN 3@16x RX: 36.33 MiB/s TX: 10.21 MiB/s
GPU 2047MHz MEM 7300MHz TEMP  56°C  FAN   0%   POW 139 / 140 W
GPU[||||||||||||||||||||||||||||||100%] MEM[||||||||           2.789Gi/12.000Gi]
```

zhonguncle — zhonguncle@zhonguncle-Precision-Tower-5810: ~ — ssh zhonguncle@169.254.2....
100
75
50
25
0
GPU0 %
GPU0 mem%
47s
35s
23s
11s
0s

```
  PID       USER DEV    TYPE  GPU      GPU  MEM    CPU  HOST MEM Command
 6428 zhonguncle   0 Compute 100%  2188MiB  18%   100%    781MiB /home/zhonguncle/miniconda3/bi
 1648 zhonguncle   0 Graphic   0%   361MiB   3%     0%     67MiB /usr/lib/xorg/Xorg vt2 -displa
 1934 zhonguncle   0 Graphic   0%    42MiB   0%     0%    451MiB /usr/bin/gnome-shell
```



```
[6]: # +100Mhz -200 140W
     import torch
     from torch.utils import benchmark
     typ = torch.float16
     n = 1024*16
     a = torch.randn(n,n).type(typ).cuda()
     b = torch.randn(n,n).type(typ).cuda()
     t = benchmark.Timer (
         stmt='a @ b',
         globals={'a':a,'b':b}
     )
     x=t.timeit(50)
     2*n**3/x.median/1e12

[6]: 28.55756119171747
```

You can see power consumption dropped by 30W while FP16 performance increased by 0.5 TFLOPS.

Let's explain voltage adjustment in detail.

Explaining `nvmlDeviceSetGpcClkVfOffset`

`nvmlDeviceSetGpcClkVfOffset(GPU, 200)` adjusts the GpcClk V/F (Voltage/Frequency) curve offset.

> GPC (Graphics Processing Cluster) is a core hardware module in modern GPU architectures that manages and schedules key computing resources. The GpcClk V/F curve is the foundation of Boost technology. When GPU load increases, the driver automatically raises GpcClk and adjusts voltage to reach Boost frequency.

If frequency is fixed, reducing this ratio lowers voltage—right?

First, we need to understand the unit of this offset value.

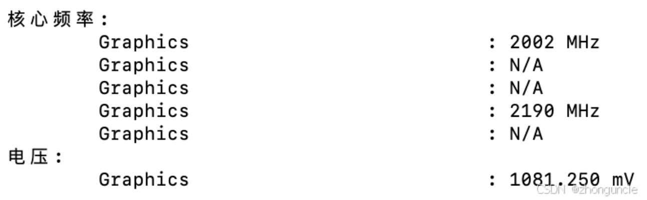
Through testing, I found this value represents a "frequency reduction". For example, setting it to `100` means the voltage used at the current frequency is the same as the default voltage at `-100 MHz`.

Let's verify this:

1. Force the frequency to 2000 MHz using the method from Section 1
2. Monitor real-time frequency and voltage in a new window:

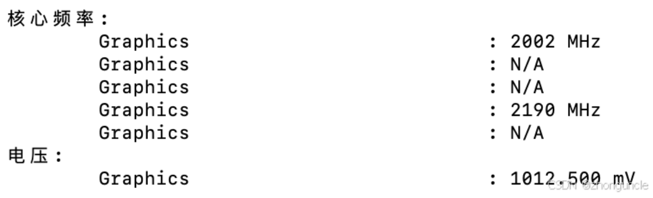
```
$ watch -n 1 'echo "Core Frequency:"; nvidia-smi -q -d CLOCK | grep "Graphics"; echo
"Voltage:"; nvidia-smi -q -d VOLTAGE | grep "Graphics"'
```

- With `nvmlDeviceSetGpcClkVfOffset(GPU, 0)`: Voltage at 2000 MHz is 1081.250 mV



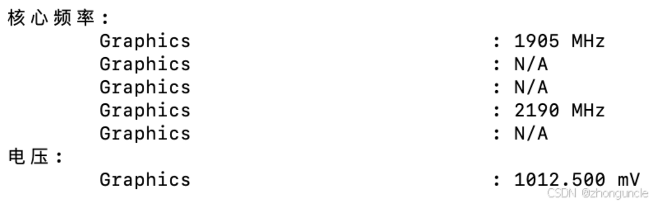
```
核心频率:
        Graphics                          : 2002 MHz
        Graphics                          : N/A
        Graphics                          : N/A
        Graphics                          : 2190 MHz
        Graphics                          : N/A
电压:
        Graphics                          : 1081.250 mV
```

- With `nvmlDeviceSetGpcClkVfOffset(GPU, 100)`: Voltage at 2000 MHz drops to 1012.500 mV



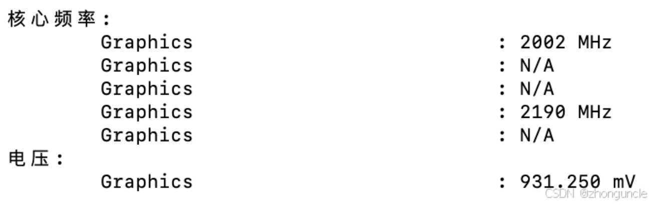
```
核心频率:
        Graphics                          : 2002 MHz
        Graphics                          : N/A
        Graphics                          : N/A
        Graphics                          : 2190 MHz
        Graphics                          : N/A
电压:
        Graphics                          : 1012.500 mV
```

- Reset offset to `0` but set frequency to 1900 MHz: Voltage is 1012.500 mV (same as above!)

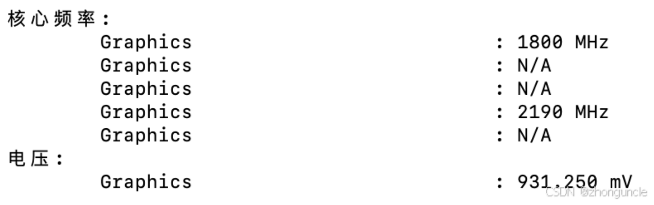


```
核心频率:
        Graphics                          : 1905 MHz
        Graphics                          : N/A
        Graphics                          : N/A
        Graphics                          : 2190 MHz
        Graphics                          : N/A
电压:
        Graphics                          : 1012.500 mV
```

- Compare `nvmlDeviceSetGpcClkVfOffset(GPU, 200)` (2000 MHz) with 1800 MHz (0 offset): Both show ~950 mV



```
核心频率:
        Graphics                          : 2002 MHz
        Graphics                          : N/A
        Graphics                          : N/A
        Graphics                          : 2190 MHz
        Graphics                          : N/A
电压:
        Graphics                          : 931.250 mV
```



```
核心频率:
        Graphics                          : 1800 MHz
        Graphics                          : N/A
        Graphics                          : N/A
        Graphics                          : 2190 MHz
        Graphics                          : N/A
电压:
        Graphics                          : 931.250 mV
```

Confirmed!

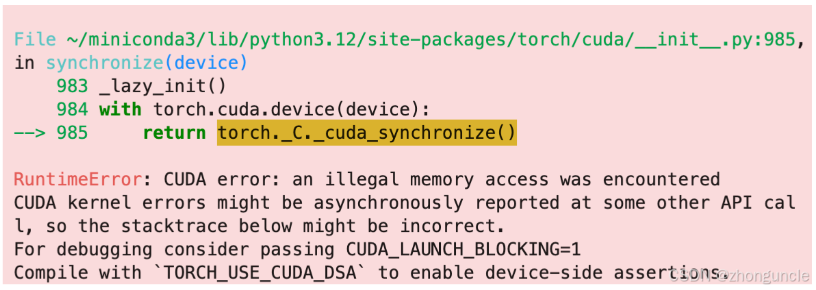
Precautions

After setting the offset, voltage is no longer constant—since frequency dynamically changes.

Even with forced frequency settings, ensure sufficient power headroom; otherwise, the GPU will throttle.

Also monitor temperature: hitting the temperature threshold will cause throttling (this threshold can be adjusted).

If you encounter errors like this during FP16 benchmarking, your settings are unstable:



```
File ~/miniconda3/lib/python3.12/site-packages/torch/cuda/__init__.py:985,
in synchronize(device)
    983 _lazy_init()
    984 with torch.cuda.device(device):
--> 985     return torch._C._cuda_synchronize()

RuntimeError: CUDA error: an illegal memory access was encountered
CUDA kernel errors might be asynchronously reported at some other API cal
l, so the stacktrace below might be incorrect.
For debugging consider passing CUDA_LAUNCH_BLOCKING=1
Compile with `TORCH_USE_CUDA_DSA` to enable device-side assertions.
```

Simply restart the Jupyter kernel (no system reboot needed). **Failing to restart will cause persistent errors regardless of subsequent adjustments.**

Adjustment Strategies

> For best results, combine the two methods described above.

First, decide whether to prioritize performance or power efficiency.

Power Efficiency (Undervolting + Overclocking)

To maintain performance while reducing power consumption:

1. Force maximum frequency near the default Boost frequency (or slightly lower)
2. Adjust `nvmlDeviceSetGpcClkVfOffset` to reduce power consumption significantly

Example for my GPU:

```
sudo nvidia-smi --id=GPU-037a6092-3f62-5b68-ac66-856918216320 -lgc 200,2100
```

Then set power limit and voltage via Python:

```
# Set power limit to 140 W (140000 mW)
nvmlDeviceSetPowerManagementLimit(GPU, 140000)
# Set GpcClk V/F curve offset (indirect voltage reduction)
nvmlDeviceSetGpcClkVfOffset(GPU, 200)
```

Setting frequencies via command line is more stable than adjusting offsets directly in Python.

This setup saves 30W of power—about 0.7 kWh per day at full load (and lowers temperatures too!). Higher-power GPUs will save even more energy.

Performance (Overclocking)

For pure performance gains:

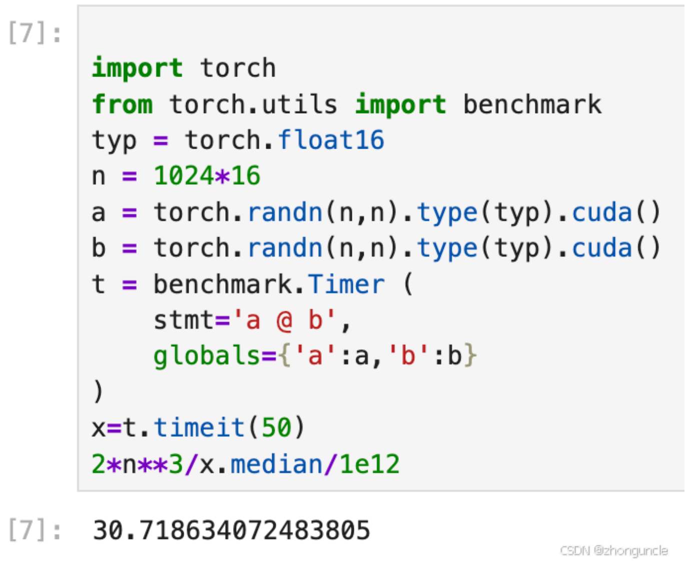
1. Expand frequency range
2. Adjust `nvmlDeviceSetGpcClkVfOffset` to optimize voltage

From the supported clocks list, my GPU's maximum frequency is 2190 MHz:

```
$ sudo nvidia-smi --id=GPU-037a6092-3f62-5b68-ac66-856918216320 -lgc 200,2190
```

Running at this frequency will hit the power limit—raise it to maximum (187W). However, even maximum power can't sustain peak frequency, so reduce voltage incrementally. Here's my stable configuration:

```
nvmlDeviceSetPowerManagementLimit(GPU, 187000)
nvmlDeviceSetGpcClkVfOffset(GPU, 170)
```



```
[7]:
import torch
from torch.utils import benchmark
typ = torch.float16
n = 1024*16
a = torch.randn(n,n).type(typ).cuda()
b = torch.randn(n,n).type(typ).cuda()
t = benchmark.Timer (
    stmt='a @ b',
    globals={'a':a,'b':b}
)
x=t.timeit(50)
2*n**3/x.median/1e12
```

```
[7]: 30.718634072483805
```

This delivers a 7.8% performance increase with a 10% (17W) power consumption increase.

I hope these tips help anyone who needs them!

Quick Reference

Command Line:

- Adjust core frequency: `sudo nvidia-smi --id=[id] -lgc [min_freq],[max_freq]`
- Adjust memory frequency: `sudo nvidia-smi --id=[id] -lmc [min_freq],[max_freq]`
- Reset core frequency: `sudo nvidia-smi --id=[id] -rgc`
- Reset memory frequency: `sudo nvidia-smi --id=[id] -rmc`
- Enable persistence mode: `sudo nvidia-smi --id=[id] -pm 1`
- Set power limit: `sudo nvidia-smi --id=[id] -pl [power_limit]`

Python:
Dynamic frequency adjustment (+100MHz), fixed power reduction (-30W), and voltage tuning:

> Still recommend reading the full explanation first!

```python
from pynvml import *
# Initialize NVML
nvmlInit()
GPU = nvmlDeviceGetHandleByIndex(0)

# Set core clock offset to +100 MHz
nvmlDeviceSetGpcClkVfOffset(GPU, 100)
# Set memory clock offset to 0 MHz
nvmlDeviceSetMemClkVfOffset(GPU, 0)
# Set power limit to 140 W (140000 mW)
nvmlDeviceSetPowerManagementLimit(GPU, 140000)
# Set GpcClk V/F (Voltage/Frequency) curve offset to -200 MHz
nvmlDeviceSetGpcClkVfOffset(GPU, 200)

# Shutdown NVML
nvmlShutdown()
```

References/Further Reading

- 5.18. Device Commands - NVML API: NVML API documentation (where I found the voltage adjustment method). Note: NVIDIA mentions deprecated APIs—my driver version may be too old for the new ones.
- How do I reset NVIDIA GPU power management settings using NVML? - Massed Compute: Learned Python-based adjustment methods here.
- ASUS V/F Point Offset - SkatterBencher: Explains ASUS CPU V/F point offset (contains useful V/F curve concepts).
- Voltage Frequency Curve - NVIDIA GeForce RTX 5080 Founders Edition Review - TechPowerUp: Shows RTX 5080 V/F curves and clock/voltage relationships.

For GPU power management research, check these papers:

- GPGPU Power Modeling for Multi-domain Voltage-Frequency Scaling
- Effects of Dynamic Voltage and Frequency Scaling on a K20 GPU
- The Impact of GPU DVFS on the Energy and Performance of Deep Learning: an Empirical Study