Final Project – Breaking the Concurrency Ceiling

Optimized Distributed Computing with Linus and CUDA MPS

David Comeau

INFT4000 – Special Topics I

Nadia Gouda

April 15, 2026

Table of Contents

Introduction 3
Background and Theory 4
  Embarrassingly Parallel Workloads 4
The WDDM Scheduling Problem 4
CUDA Multi-Process Service 5
Exploration and Implementation 6
  Step One – Dual-Boot Linux Mint Installation 6
  Step Two – NVIDIA Driver and CUDA Toolkit 7
  Step Three – MPS Daemon Configuration 8
  Step Four - GPU Lock Configuration 10
  Step Five – BOINC Client and app_config.xml 11
  Step Six – Verification 11
Results and Findings 13
Technology Evaluation 15
Reflection 16
References 18

Introduction

This semester began with a straightforward objective: to move beyond the passive role of a volunteer in the Einstein@Home distributed computing network and become an active technical analyst. In Assignment Four, I defined three learning goals - mastering advanced client configuration, analyzing hardware efficiency, and building programmatic log analysis tools. What I did not anticipate was that pursuing those goals would lead me to fundamentally reconsider a conclusion I had published in a previous assignment, transition my primary workstation to a new operating system, and engage directly with the global volunteer computing community to solve a problem I could not crack on my own.

The trajectory of this exploration followed a clear arc. In Assignments Five and Six, I demonstrated that overriding the default BOINC client behavior with a custom app_config.xml file could force concurrent GPU tasks, effectively doubling throughput on my RTX 3090 for a minimal increase in power consumption. In Assignment Nine, after upgrading to an RTX 5090, I tested whether this concurrency could scale further. It could not. Pushing beyond two concurrent tasks on Windows caused completion times to balloon, and I concluded that the GPU's memory bandwidth and cache architecture imposed a hard physical ceiling on local parallelism.

That conclusion was wrong. This final project documents how I discovered, implemented, and validated the real solution: NVIDIA's CUDA Multi-Process Service (MPS), a Linux-exclusive technology that eliminates the artificial scheduling bottleneck imposed by the Windows display driver model. The investigation was triggered by a direct conversation with another Einstein@Home volunteer, Petri33, who achieves a Recent Average Credit of approximately 9.5 million on identical hardware - nearly four times my Windows output.

Background and Theory

Embarrassingly Parallel Workloads

As established in Assignment Three, the computational foundation of the Einstein@Home network relies on "embarrassingly parallel" workloads. These are datasets that can be separated into independent fragments requiring no communication or data dependency between them. In the context of the All-Sky Gravitational Wave Search (O4), each work unit applies a matched filtering template over a discrete segment of LIGO detector data to search for the continuous gravitational wave signature of a spinning neutron star. Because one frequency band does not depend on the results of another, the central server can distribute millions of these units to volunteers worldwide for simultaneous processing.

This theoretical independence is what makes local GPU concurrency possible in the first place. If two work units shared data or required synchronization, running them simultaneously on the same GPU would risk data collisions or calculation errors. Because they are mathematically isolated, the only constraints on concurrency are the physical resources of the hardware and the efficiency of the software scheduling the work.

The WDDM Scheduling Problem

On Windows, NVIDIA GPUs operate under the Windows Display Driver Model (WDDM). WDDM was designed for desktop display compositing, not high-performance compute. Every application that touches the GPU - browser windows, the desktop compositor (DWM), video playback, and CUDA compute tasks - passes through the same driver scheduling layer. When multiple CUDA processes run simultaneously, WDDM time-slices between them, granting each process exclusive but temporary access to the GPU. The processes do not share the hardware; they take turns.

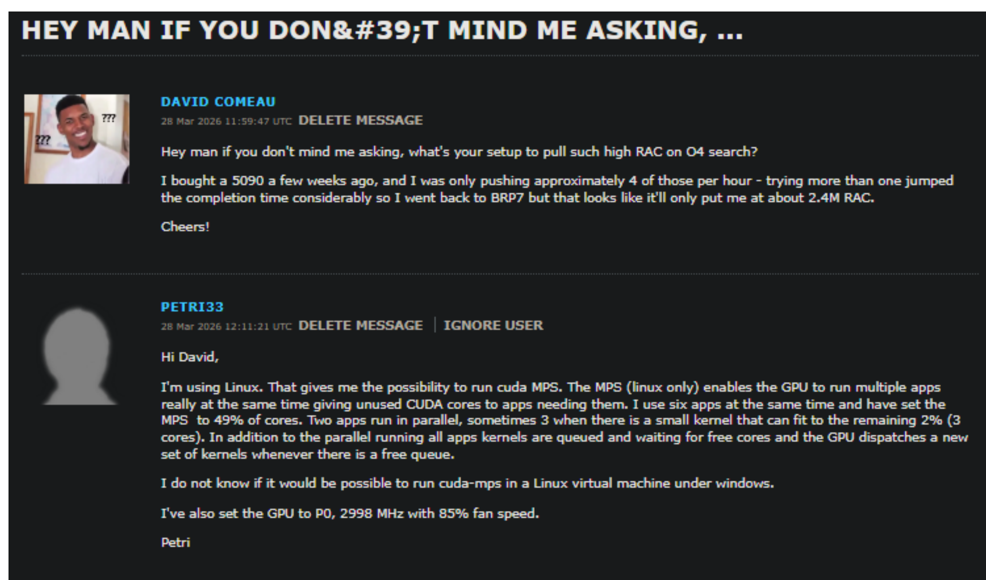
This time-slicing is why my Assignment Nine testing showed catastrophic performance degradation beyond two concurrent tasks. Each additional CUDA context added scheduling overhead and forced the GPU to repeatedly flush and reload state between processes. The tasks were not contending for compute resources - they were contending for the driver's attention. Even minor GPU activity from non-compute sources, such as a web browser with hardware acceleration enabled, introduced measurable slowdowns. During my Windows testing, simply watching a YouTube video added 10 to 15 seconds to each task's completion time, because the browser's GPU compositor was forcing additional context switches.

CUDA Multi-Process Service

CUDA MPS is a Linux-exclusive runtime service that fundamentally changes how multiple CUDA processes interact with the GPU. Instead of each process creating its own isolated GPU context and time-slicing for access, MPS creates a single shared context managed by a dedicated server process. All client processes submit their CUDA kernels through this shared context, and the GPU's hardware scheduler dispatches them to available Streaming Multiprocessors (SMs) simultaneously. The result is genuine parallel execution: multiple processes run on the GPU at the same time, on different SMs, without time-slicing.

A critical configuration parameter is CUDA_MPS_ACTIVE_THREAD_PERCENTAGE, which controls the maximum portion of the GPU's SMs that any single client process can occupy. Setting this to 49% means each task can use up to 49% of the SMs. With two tasks active, 98% of the SMs are occupied, and a third task can occasionally fit smaller kernels into the remaining 2%. Additional queued tasks have their kernels staged and ready for immediate dispatch whenever SMs become available, creating a pipeline that ensures the GPU never sits idle between kernel completions.

This is the configuration described by Petri33 in his message: six tasks loaded in BOINC, with two running in true parallel at any given moment and the rest queued for immediate dispatch. The key insight is that MPS is completely transparent to the applications - BOINC and the Einstein@Home compute binaries require no modification whatsoever. The parallelism is handled entirely at the driver level.



HEY MAN IF YOU DON'T MIND ME ASKING, ...

DAVID COMEAU
28 Mar 2026 11:59:47 UTC DELETE MESSAGE

Hey man if you don't mind me asking, what's your setup to pull such high RAC on O4 search?

I bought a 5090 a few weeks ago, and I was only pushing approximately 4 of those per hour - trying more than one jumped the completion time considerably so I went back to BRP7 but that looks like it'll only put me at about 2.4M RAC.

Cheers!

PETRI33
28 Mar 2026 12:11:21 UTC DELETE MESSAGE | IGNORE USER

Hi David,

I'm using Linux. That gives me the possibility to run cuda MPS. The MPS (linux only) enables the GPU to run multiple apps really at the same time giving unused CUDA cores to apps needing them. I use six apps at the same time and have set the MPS to 49% of cores. Two apps run in parallel, sometimes 3 when there is a small kernel that can fit to the remaining 2% (3 cores). In addition to the parallel running all apps kernels are queued and waiting for free cores and the GPU dispatches a new set of kernels whenever there is a free queue.

I do not know if it would be possible to run cuda-mps in a Linux virtual machine under windows.

I've also set the GPU to P0, 2998 MHz with 85% fan speed.

Petri

Figure 1 - Petri33's private message describing his MPS configuration.

Exploration and Implementation

The implementation was conducted on the following hardware: an AMD Ryzen 9 9950X3D processor (16 cores, 32 threads, 96MB V-Cache), an ASUS ROG Astral GeForce RTX 5090 LC OC (32GB GDDR7, liquid cooled), and 64GB of DDR5-6000 memory, installed on an ASRock X670E Taichi Carrara motherboard.

Step One – Dual-Boot Linux Mint Installation

I installed Linux Mint 22 Cinnamon Edition alongside my existing Windows 11 Professional installation. A known issue with dual-boot configurations is that Windows and Linux handle the hardware clock differently: Windows assumes the system clock stores local time, while Linux assumes it stores UTC. Without correction, switching between operating systems causes the clock to be offset by the local time zone difference. I resolved this by applying a registry modification on the Windows side, setting the RealTimeIsUniversal DWORD value to 1 in the TimeZoneInformation key. This instructs Windows to treat the hardware clock as UTC, matching Linux's default behavior.

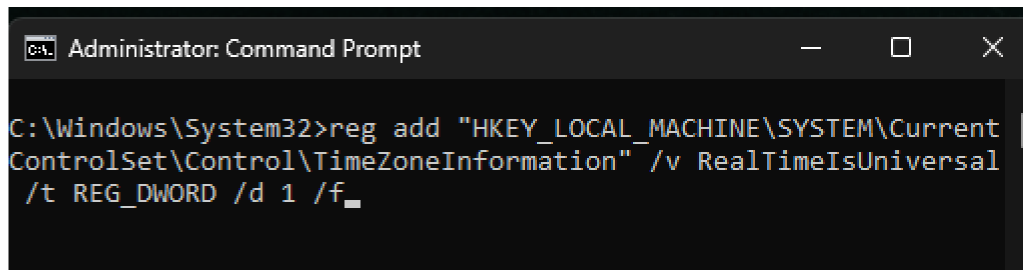


Figure 2 - Windows registry fix for RealTimeIsUniversal.

Step Two – NVIDIA Driver and CUDA Toolkit

The RTX 5090, built on NVIDIA's Blackwell architecture, requires the open kernel modules. The fully proprietary NVIDIA driver does not recognize this GPU and will fail with a "No devices were found" error. After booting into Linux Mint on the generic nouveau display driver, I added the graphics-drivers PPA repository to access up-to-date driver packages. Linux Mint's Driver Manager then displayed the available open drivers. I selected nvidia-driver-590-open (version 590.48.01), which installed the open kernel modules and the full CUDA driver stack.

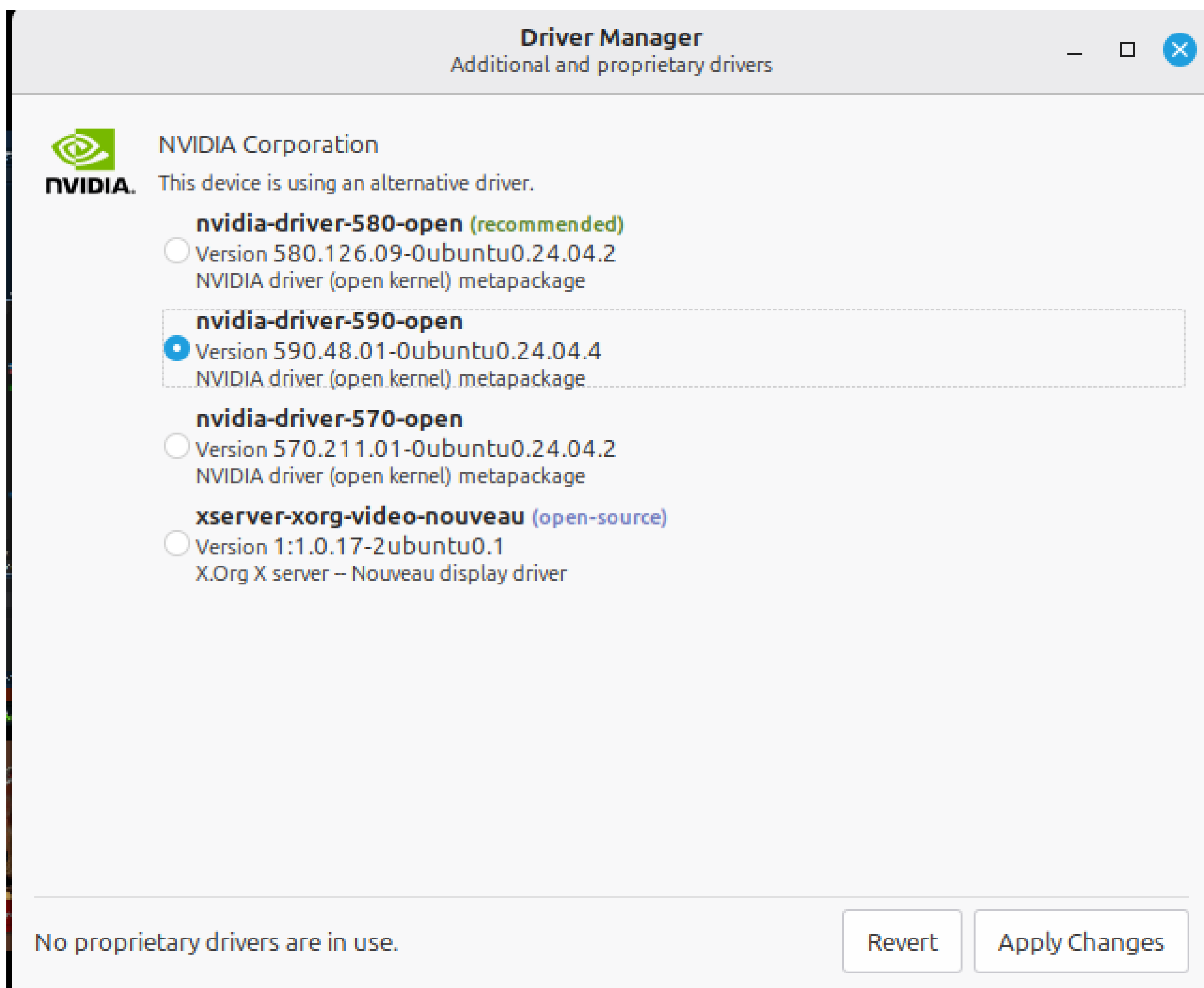


Figure 3 - Driver Manager showing nvidia-driver-590-open selected.

After rebooting, I verified the installation using nvidia-smi, which confirmed the RTX 5090 was recognized with driver version 590.48.01 and CUDA version 13.1. I then installed the CUDA toolkit via the nvidia-cuda-toolkit package, which provides the nvidia-cuda-mps-control binary required for MPS.

```
dj@Woodsmoke: ~
dj@Woodsmoke:~$ nvidia-smi
Tue Apr 14 22:50:21 2026
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 590.48.01              Driver Version: 590.48.01      CUDA Version: 13.1     |
+-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 5090        Off |   00000000:01:00.0  On |                  N/A |
|  0%   36C    P3             35W /  600W |     537MiB /  32607MiB |      1%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI              PID   Type   Process name                        GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|    0   N/A  N/A            1563      G   /usr/lib/xorg/Xorg                      233MiB |
|    0   N/A  N/A            1980      G   cinnamon                                 18MiB |
|    0   N/A  N/A           13432      G   /usr/lib/firefox/firefox                222MiB |
+-----------------------------------------------------------------------------------------+
```

Figure 4 - nvidia-smi output confirming RTX 5090 and driver version.

Step Three – MPS Daemon Configuration

CUDA MPS runs as a background daemon. To ensure it starts automatically on every boot, I created a systemd service file at /etc/systemd/system/nvidia-mps.service. The service sets the CUDA_MPS_ACTIVE_THREAD_PERCENTAGE environment variable to 49 before launching the MPS control daemon, replicating Petri33's configuration. The service also enables NVIDIA persistence mode, which keeps the driver loaded between compute sessions and prevents the relatively long initialization delay that occurs when a CUDA application starts on an unloaded driver.

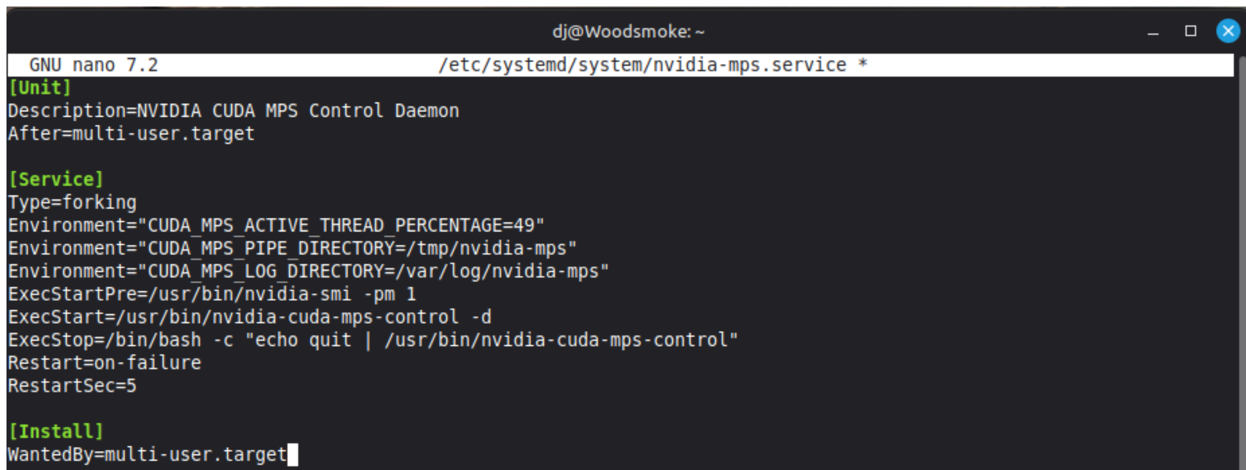


Figure 5 - nvidia-mps-service file contents in nano

After enabling and starting the service, I verified that MPS was operational by querying the default active thread percentage. The daemon returned 49.0, confirming the configuration was active.

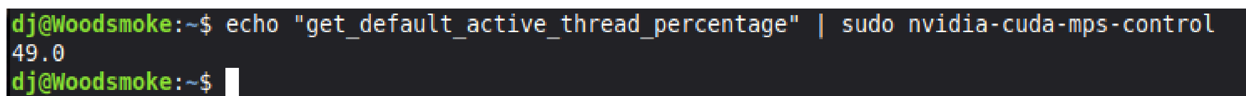


Figure 6 - systemctl status showing MPS active; thread percentage query returning 49.0.

Step Four - GPU Lock Configuration

Petri33 locks his GPU to P0 performance state at 2998 MHz. Locking the clocks prevents the GPU from dynamically downclocking between kernel dispatches, which reduces latency and ensures consistent throughput. I created a second systemd service at /etc/systemd/system/nvidia-gpu-config.service that runs nvidia-smi -lgc 2998,2998 after the MPS service starts. An initial attempt to make this service depend on nvidia-persistenced.service failed because that service was not present in my configuration. Changing the dependency to nvidia-mps.service resolved the issue.

```
dj@Woodsmoke:~$ sudo systemctl daemon-reload
[sudo] password for dj:
dj@Woodsmoke:~$ sudo systemctl enable nvidia-gpu-config.service
Created symlink /etc/systemd/system/multi-user.target.wants/nvidia-gpu-config.service → /etc/systemd/system/nvidia
-gpu-config.service.
dj@Woodsmoke:~$ sudo systemctl start nvidia-gpu-config.service
dj@Woodsmoke:~$ sudo systemctl status nvidia-gpu-config.service
● nvidia-gpu-config.service - NVIDIA GPU Clock and Power Configuration
     Loaded: loaded (/etc/systemd/system/nvidia-gpu-config.service; enabled; preset: enabled)
     Active: active (exited) since Tue 2026-04-14 23:19:52 ADT; 4s ago
    Process: 42448 ExecStart=/usr/bin/nvidia-smi -pm 1 (code=exited, status=0/SUCCESS)
    Process: 42450 ExecStart=/usr/bin/nvidia-smi -lgc 2998,2998 (code=exited, status=0/SUCCESS)
   Main PID: 42450 (code=exited, status=0/SUCCESS)
        CPU: 29ms

Apr 14 23:19:52 Woodsmoke systemd[1]: Starting nvidia-gpu-config.service - NVIDIA GPU Clock and Power Configurati>
Apr 14 23:19:52 Woodsmoke nvidia-smi[42448]: Persistence mode is already Enabled for GPU 00000000:01:00.0.
Apr 14 23:19:52 Woodsmoke nvidia-smi[42448]: All done.
Apr 14 23:19:52 Woodsmoke nvidia-smi[42450]: GPU clocks set to "(gpuClkMin 2998, gpuClkMax 2998)" for GPU 0000000>
Apr 14 23:19:52 Woodsmoke nvidia-smi[42450]: All done.
Apr 14 23:19:52 Woodsmoke systemd[1]: Finished nvidia-gpu-config.service - NVIDIA GPU Clock and Power Configurati>
```

Figure 7 - nvidia-gpu-config.service status showing clocks locked successfully.

Step Five – BOINC Client and app_config.xml

I installed the BOINC client and manager packages and attached to the Einstein@Home project using my existing account. After allowing the client to synchronize with the project and download its first batch of tasks, I identified the Linux application name for the O4 gravitational wave search by inspecting the job log file. The application is registered as einstein_O4AS on Linux, which differs from the Windows identifier used in my previous assignments.

I created an app_config.xml file in the Einstein@Home project directory with gpu_usage set to 0.50, instructing the BOINC scheduler to treat each task as requiring half of the GPU. This allows two tasks to run concurrently, aligning with the MPS thread allocation of 49% per task.

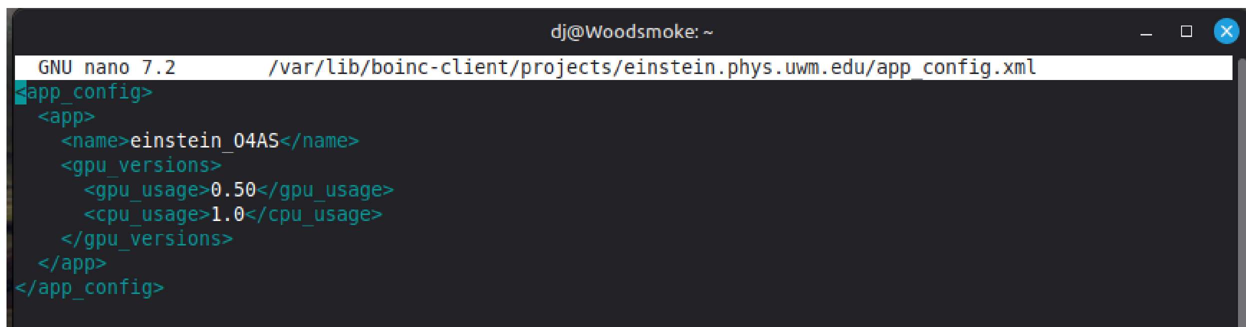


Figure 8 - app_config.xml contents in nano.

Step Six – Verification

With all components in place, I verified the complete stack. The nvidia-smi output showed the nvidia-cuda-mps-server process managing two concurrent gw-linux-gpu compute tasks, GPU utilization at approximately 100%, performance state P0, and no thermal or power throttling. The BOINC Manager confirmed two O4 tasks running simultaneously with the status "Running (1 CPU + 0.5 NVIDIA GPUs)."

```
dj@Woodsmoke:~$ nvidia-smi
Tue Apr 14 23:32:33 2026
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 590.48.01              Driver Version: 590.48.01      CUDA Version: 13.1     |
+-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 5090        On  |   00000000:01:00.0  On |                  N/A |
| 30%   50C    P0            316W /  600W |    4292MiB /  32607MiB |     88%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI              PID   Type   Process name                        GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|    0   N/A  N/A            1563      G   /usr/lib/xorg/Xorg                       275MiB |
|    0   N/A  N/A            1980      G   cinnamon                                  23MiB |
|    0   N/A  N/A           13432      G   /usr/lib/firefox/firefox                 228MiB |
|    0   N/A  N/A           47511      C   nvidia-cuda-mps-server                    52MiB |
|    0   N/A  N/A           48473      G   boincmgr                                   6MiB |
|    0   N/A  N/A           50387    M+C   ...-pc-linux-gnu__GW-cuda-580-2G        1812MiB |
|    0   N/A  N/A           52298    M+C   ...-pc-linux-gnu__GW-cuda-580-2G        1812MiB |
+-----------------------------------------------------------------------------------------+
```

Figure 9 - nvidia-smi showing MPS server and two concurrent compute tasks.

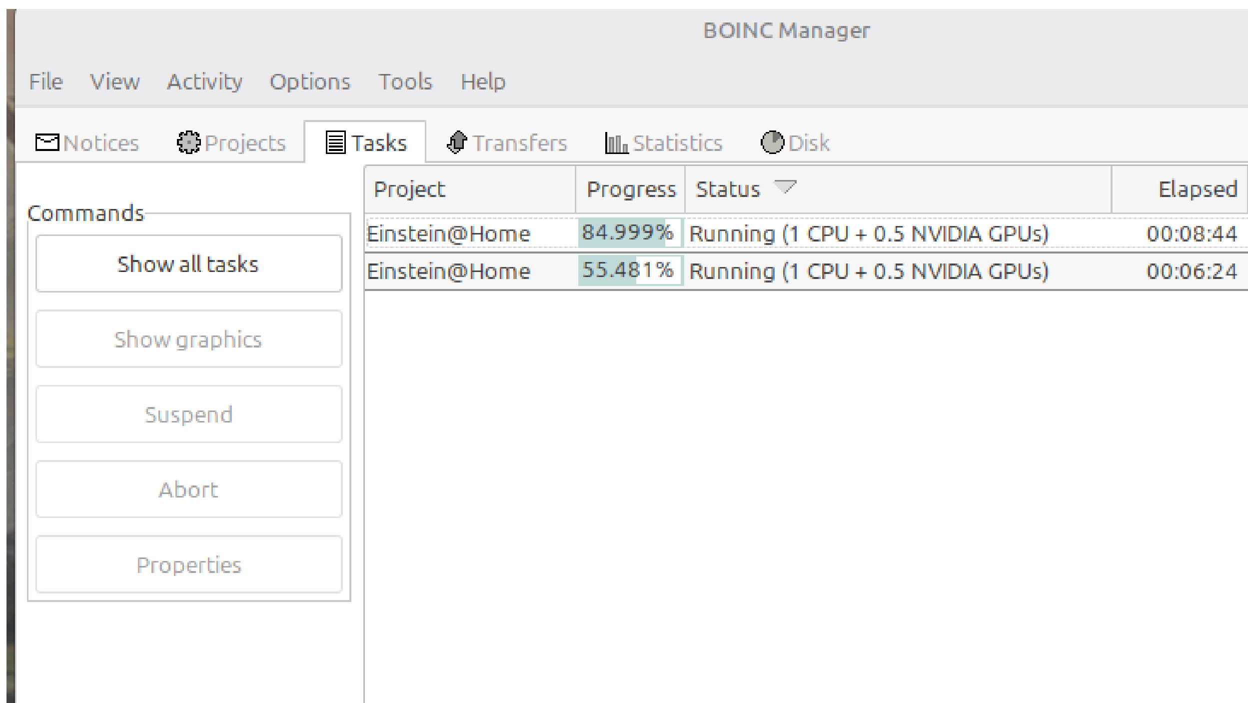


Figure 10 - BOINC Manager showing two O4 tasks running simultaneously.

Results and Findings

The transition from Windows with standard CUDA scheduling to Linux with MPS produced a dramatic improvement in throughput for the All-Sky Gravitational Wave Search tasks.

Table 1: Performance Comparison Across Configurations

Configuration	Concurrent Tasks	Avg. Time per Task	Credit per Task	Credit/sec	Est. Daily RAC
Win – 1 Task (BRP7)	1	~2:00	3,333	27.8	~2.4M
Win – 2 Tasks (BRP7)	2	~3:50	3,333	28.9	~2.5M
Win – 1 Task (O4)	1	~15:00	18,000	20.0	~1.7M
Win – 2 Tasks (O4)	2	~20:00	18,000	30.0	~2.6M
Linux + MPS (O4)	2	~8:25	18,000	71.3	~6.2M
Petri33 (Reference)	6 (2 parallel)	~12:55	18,000	~110	~9.5M

h1_2232.60_O4a12 00C00Cl1In0__O4A SUHF_2233.00Hz_1 1264_0	1004592745	15 Apr 2026 6:39:58 UTC	15 Apr 2026 8:39:21 UTC	Completed, waiting for validation	503	501	0	All-Sky Gravitational Wave search on O4 v2.09 (GW-cuda-580-2G-B) x86_64-pc-linux-gnu
h1_2232.60_O4a12 00C00Cl1In0__O4A SUHF_2233.00Hz_1 1263_0	1004592748	15 Apr 2026 6:40:00 UTC	15 Apr 2026 8:22:38 UTC	Completed, waiting for validation	503	501	0	All-Sky Gravitational Wave search on O4 v2.09 (GW-cuda-580-2G-B) x86_64-pc-linux-gnu
h1_2232.60_O4a12 00C00Cl1In0__O4A SUHF_2233.00Hz_1 1262_0	1004592751	15 Apr 2026 6:40:02 UTC	15 Apr 2026 8:39:21 UTC	Completed, waiting for validation	504	502	0	All-Sky Gravitational Wave search on O4 v2.09 (GW-cuda-580-2G-B) x86_64-pc-linux-gnu
h1_2232.60_O4a12 00C00Cl1In0__O4A SUHF_2233.00Hz_1 1260_0	1004594108	15 Apr 2026 6:52:41 UTC	15 Apr 2026 8:39:21 UTC	Completed, waiting for validation	503	501	0	All-Sky Gravitational Wave search on O4 v2.09 (GW-cuda-580-2G-B) x86_64-pc-linux-gnu

« FIRST ‹ PREVIOUS 1 2 3 4

Figure 11 - Completed task batch showing elapsed times.

The most significant result is the per-task completion time for the O4 gravitational wave search. On Windows, a single O4 task completed in approximately 15 minutes. Attempting two concurrent tasks on Windows caused time-slicing overhead that stretched individual task times to roughly 20 minutes, producing only a marginal throughput increase. On Linux with MPS, two tasks completed in approximately 8 minutes and 25 seconds each, running in true parallel. This is faster per task than a single task on Windows, while processing twice the workload.

The effective throughput jumped from approximately 20 credit per second on Windows to 71.3 credit per second on Linux with MPS - a 3.5x improvement using the same GPU. This conclusively demonstrates that the concurrency ceiling I identified in Assignment Nine was not imposed by the GPU's physical architecture, but by the Windows WDDM driver's scheduling model.

An additional finding during testing was that browser activity with hardware acceleration enabled introduced measurable performance degradation, even on Linux. Firefox's GPU-accelerated compositor consumed VRAM and forced context switches, reducing GPU utilization from approximately 100% to 73% and visibly slowing task completion. Disabling hardware acceleration in Firefox or closing the browser entirely restored full performance. This mirrors the YouTube observation on Windows and reinforces that any GPU consumer - not just CUDA compute - impacts distributed computing throughput.

My current projected daily RAC of approximately 6.2 million is still short of Petri33's 9.5 million. The gap is primarily attributable to queue depth: he runs six tasks to keep the MPS kernel dispatch pipeline fully saturated, while my testing used two. Additionally, my per-task time of approximately 505 seconds at two concurrent is substantially faster than his reported times of approximately 775 seconds, likely due to the 9950X3D's superior single-threaded performance on the CPU-bound portions of the O4 workload. This suggests that increasing the queue depth while maintaining low per-task times could potentially match or exceed his throughput.

Technology Evaluation

CUDA Multi-Process Service is a narrowly focused but exceptionally effective technology for a specific class of GPU workloads. The following evaluation considers its strengths, limitations, and appropriate use cases within the context of distributed volunteer computing.

Table 2: Technology Evaluation – CUDA MPS

Metric	Assessment
Strengths	Completely transparent to applications - no code changes required. Enables genuine parallel kernel execution across SMs rather than time-slicing. Configurable thread allocation allows fine-tuning per workload. Runs as a standard systemd service with minimal administrative overhead. Eliminates the WDDM scheduling bottleneck entirely.
Limitations	Linux-only - not available on Windows under any configuration, including WSL2. Requires careful tuning of the active thread percentage; incorrect values degrade rather than improve performance. Fault isolation is limited - a fatal GPU error in one MPS client can affect the MPS server and all connected clients. The technology is poorly documented for consumer use cases; virtually all NVIDIA documentation targets HPC and data center deployments.
Use Cases	Ideal for multi-process GPU compute workloads where individual tasks do not fully saturate the GPU's streaming multiprocessors. Distributed computing projects like BOINC are a textbook application. Also applicable to multi-model AI inference serving, where multiple small models can share a GPU more efficiently than time-slicing. Not suitable for workloads that individually saturate the GPU, such as large-scale neural network training.

When compared against the alternative - running concurrent CUDA tasks without MPS on either Windows or Linux - MPS is categorically superior for this workload. Without MPS, the GPU time-slices between contexts regardless of operating system. MPS is the only mechanism that enables true parallel execution of independent CUDA processes on a single GPU. For the specific use case of distributed volunteer computing with embarrassingly parallel workloads, there is no viable alternative that achieves equivalent throughput.

Reflection

When I began this course in January, I described myself in Assignment One as someone who was "comfortable diving into unfamiliar tech and experimenting until I find a solution." I identified self-motivation as my primary strength and time management as my primary challenge. Both of those assessments held true, but the nature of the experimentation and the scale of the solutions evolved far beyond what I initially anticipated.

My understanding of hardware optimization underwent a fundamental correction over the course of the semester. In Assignment Nine, I concluded with confidence that pushing beyond two concurrent GPU tasks saturated the RTX 5090's memory bandwidth and cache limits - a physical, architectural limitation of the silicon itself. I wrote that "even if a mathematical problem is perfectly separable into independent work units, the hardware executing those units - specifically the memory controllers and cache bandwidth – acts as the ultimate ceiling for efficiency." That conclusion was wrong. The ceiling was not in the hardware. It was in the software layer sitting between the application and the hardware: the Windows display driver model. Discovering this through the implementation of MPS was the single most important technical insight of the semester. It taught me that diagnosing a performance bottleneck requires examining the entire stack - from the application, through the driver, to the hardware - and that a plausible-sounding explanation at one layer can mask the true cause at another.

The most practically valuable skill I developed was Linux systems administration. Configuring systemd services, managing environment variables for daemon processes, debugging service dependency failures, installing and managing GPU drivers from PPA repositories, and working with the NVIDIA driver stack on Linux are all skills that translate directly to professional infrastructure and DevOps work. Before this project, my Linux experience was limited to casual desktop use. After it, I am comfortable building and managing persistent compute services on a Linux workstation.

Perhaps the most important lesson, however, was not technical at all. The entire pivot to Linux and MPS - the discovery that transformed my understanding and tripled my throughput - came from a single private message to another volunteer on the Einstein@Home forum. No amount of independent research, documentation review, or experimentation had surfaced CUDA MPS as a solution to my concurrency problem. It took a direct conversation with someone who had already solved it. In a self-directed

course built around independent learning, the most impactful moment came from asking someone else for help. That is a lesson worth remembering.

If I were to continue this exploration, my next steps would be to optimize the MPS thread percentage and task queue depth to find the absolute peak throughput for my hardware, to investigate hybrid GPU and CPU workloads that maximize the contribution of the entire system, and to analyze the power efficiency of different configurations by calculating credits per watt – balancing scientific output against the real cost of residential electricity.

References

Anderson, D. (2025, 11 7). *BOINC Wiki*. Retrieved from GitHub: https://github.com/BOINC/boinc/wiki/Client-configuration#project-level-configuration

Anderson, D. P. (2004). *Fifth IEEE/ACM International Workshop on Grid Computing, 4-10*. Retrieved from BOINC: A system for public-resource computing and storage.: https://doi.org/10.1109/GRID.2004.14

Anderson, D. P. (2019, 11 16). *BOINC: A Platform for Volunteer Computing*. Retrieved from Springer Nature Link: https://link.springer.com/article/10.1007/s10723-019-09497-9

ASUS. (n.d.). *GPU Tweak III | Ultimate GPU Tuning Tool*. Retrieved from asus.com/campaign/GPU-Tweak-III/

Caltech. (n.d.). *LIGO - A Gravitational Wave Observatory*. Retrieved from LIGO Caltech: https://www.ligo.caltech.edu/page/ligo-gw-interferometer

CPUID. (n.d.). *HWMONITOR | Softwares - CPUID.* Retrieved from https://www.cpuid.com/softwares/hwmonitor.html

GWOSC. (n.d.). Retrieved from Gravitational Wave Open Science Center: https://gwosc.org/

Knispel, B. A. (2010). *Pulsar discovery by global volunteer computing. Science, 329(5997), 1305.* Retrieved from https://www.science.org/doi/10.1126/science.1195253

McGloughlin, B. M.-B. (2026, 01 20). *Einstein@Home All-sky "Bucket" Search for Continuous Gravitational Waves in LIGO O3a Public Data*. Retrieved from The Astrophysical Journal, 997:149: https://iopscience.iop.org/article/10.3847/1538-4357/ae225a

NVidia. (2025). *Multi-Process Service*. Retrieved from NVidia: https://docs.nvidia.com/deploy/mps/index.html

NVIDIA. (2025). *ROG Astral LC GeForce RTX 5090*. Retrieved from NVIDIA: https://rog.asus.com/graphics-cards/graphics-cards/rog-astral/rog-astral-lc-rtx5090-o32g-gaming/

NVIDIA. (n.d.). *CUDA Programming Guide*. Retrieved from NVIDIA: https://docs.nvidia.com/cuda/cuda-programming-guide/

NVidia. (n.d.). *nvidia-cuda-mps-control Manual Page*. Retrieved from https://manpages.ubuntu.com/manpages/jammy/man1/nvidia-cuda-mps-control.1.html

Petri33. (2026, March 29). Personal communication via Einstein@Home Private Message.

Reinhard Prix, B. M. (2020). *Einstein@Home – gravitational waves for everybody*. Retrieved from https://www.einstein-online.info/en/spotlight/eah/