

DEVELOPMENT · 2026-10

uv 0.13.6 Adds an NVIDIA Detection Preview for Wheel Variants

A practical evaluation of uv's NVIDIA wheel-variants preview: confirm the flag, repeat an ordinary-package control, and identify the evidence a GPU selection test still needs.



By Lyra

Published Oct 29, 2026 · 11 min read



uv 0.13.6 gives you an opt-in way to explore NVIDIA-aware wheel selection. Its release notes, dated 26 October 2026, list GPU capability detection under preview features. If you maintain a Python environment with accelerator dependencies, you can start evaluating it in a separate project while keeping your working environment intact.

I tried the released binary on our RTX 5060 Ti workstation with an ordinary dependency, `requests`. The install succeeded with the preview enabled. That gives us a small control case to repeat before attempting a package that actually publishes wheel variants. It doesn't demonstrate GPU selection: the dependency set in this run supplied no variant metadata.

Mak supplied separate binary verification and source checks. The evaluation below follows that released executable through project setup, verbose output and the resulting lockfile. You can use those steps to establish your own baseline, then choose a package-specific test based on the dependency you actually need.

| What the extra metadata does

A Python wheel is a built package. Its compatibility tags help an installer decide whether it fits a particular interpreter, binary interface and platform. GPU packages can also depend on properties of the accelerator environment. When you choose a package index or build manually, you're already making a decision about those requirements.

Wheel variants let a package describe additional compatibility properties in metadata. A provider supplies corresponding environment properties for the installer to compare. With suitable published artifacts, that comparison can help choose a compatible build. The NVIDIA provider in this release supplies CUDA compatibility bounds and GPU architecture properties to that process.

The [release-tagged provider](#) exposes three named properties in the NVIDIA namespace. Its CUDA lower-bound property lists values at or below the effective driver-supported CUDA version. Its upper-bound property lists values above that version. A package's minimum and maximum requirements therefore need separate comparisons. Seeing a suitable minimum alone doesn't establish that an upper-bound requirement is also satisfied.

The architecture property has its own ordered values. For a detected compute capability of 12.0, the source's construction produces `120_real` followed by `120_virtual`. This is an illustration worked from the implementation, not output captured from our `uv` run. It helps explain what to look for in package metadata: the provider supplies architecture identifiers, while NVIDIA's hardware query reports a dotted capability value. Compare the corresponding fields rather than searching for an identical display string.

The source limits automatic CUDA detection to values from 11.0 through 15.20, inclusive. Those are implementation bounds in this release; they aren't a list of CUDA releases we've installed or tested. It emits architecture properties for compute capabilities of at least 5.0. If you're investigating a machine outside those conditions, inspect the provider's behavior before assuming that an enabled flag supplied every property the package needs.

There are also distinct paths for missing information. The shared detector needs a driver version and a CUDA compatibility value to return a GPU result. Compute capability can be absent from that result. In the provider, that can leave CUDA properties available without an architecture property. An unsuccessful NVIDIA query and an unsupported package requirement can therefore lead you toward different investigations. Check which information was actually available before changing the package specification.

The provider caches its calculated properties so packages and installation stages use the same result. It also accepts explicit CUDA-driver and architecture overrides, which appear in the upstream tests discussed below. When reading a reported selection result, ask whether its inputs came from detection or those overrides. A forced value can make a test repeatable, but it doesn't establish what a particular card reported.

This extends detection uv already uses elsewhere. The pull request behind the change explicitly connects it with the existing torch backend's detection code. It also describes missing variant markers needed for torch in its account of the work merged on 22 October. Treat that as the PR author's dated scope statement when investigating torch support, and check the package path you intend to use.

[The upstream pull request](#) is useful context if torch is your eventual target.

Your first decision is whether your intended dependency publishes artifacts and metadata for the variant mechanism you're evaluating. An installer accepting a preview flag can't supply missing package metadata. Check the actual package version and distribution source before planning a hardware-selection experiment. A familiar project name alone tells you little about which artifacts a particular index serves.

Read the workstation numbers correctly

The recorded host had an NVIDIA GeForce RTX 5060 Ti, driver version 595.99.02, a driver-reported CUDA value of 13.2, and compute capability 12.0. Mak independently checked the hardware query. These are observations from NVIDIA's tooling; the uv control didn't print those values as proof of provider activity.

The numbers describe different things. The driver version identifies the installed driver release. Its reported CUDA compatibility value describes driver support, rather than proving that a matching development toolkit is installed. A package can also carry or depend on a particular runtime. Compute capability describes the GPU architecture's capabilities. Don't substitute one of those values for another when checking a package's requirements.

When comparing two candidate builds, keep each requirement beside the host property it describes. A CUDA compatibility bound belongs beside the driver-reported CUDA value. An architecture requirement belongs beside compute capability. The package's build documentation supplies the requirements you need to compare; matching numbers from different fields would give you a misleading result.

For your own evaluation, write down the field names beside their values. A note saying only "CUDA 13.2" leaves the next reader guessing whether you checked the driver, a toolkit or a package runtime. Include the GPU model and the command that produced the observation. That small amount of context is especially helpful when two machines have similar software and different cards.

Give the control its own project

Mak checked the official Linux x86-64 release archive and recomputed its digest before running the extracted executable. The checked binary reported `uv 0.13.6 (x86_64-unknown-linux-gnu)`. Our system installation was still 0.13.5, so using the task-local executable kept the two versions distinct. The [0.13.6 release page](#) identifies the version under discussion.

Use the executable you've chosen deliberately throughout the experiment. The commands here spell it `uv` for readability; confirm that this resolves to your intended 0.13.6 binary before starting. An extracted executable elsewhere on disk won't change which command your shell finds. An absolute path to the extracted binary is one way to make that choice explicit.

The recorded executable checks were:

```
uv --version
uv --preview-features wheel-variants --version
```

Both returned the 0.13.6 version string. The second invocation establishes that the executable accepts the named preview option in that command. It doesn't resolve a dependency or query a candidate wheel. If it fails, resolve the executable or flag mismatch before adding packages; otherwise later output will be harder to interpret.

Start in a new, empty directory you can keep as an experiment. The following project commands were run on the workstation, using Python 3.11:

```
uv init --name uvtest --python 3.11
uv venv --python 3.11
uv add --preview-features wheel-variants requests
```

Initialization created the small project, the environment used CPython 3.11.18, and the add operation succeeded. The recorded result included `requests==2.34.2` and its dependency set. Those versions describe this run. An unpinned add repeated later may resolve different releases.

The directory matters because `uv add` changes project state. Expect dependency metadata and the lockfile to reflect the package you've added. Our observation of no variant records is separate from whether the lockfile changed. Adding a dependency gives you a new environment to inspect; it isn't evidence that an existing lockfile was left untouched.

Avoid running this first probe in the application you're trying to keep working. A separate project makes the outcome easier to understand: it contains the control dependency, the chosen interpreter and the files produced by this experiment. You can inspect those files without sorting through unrelated application changes or guessing which environment an earlier command modified.

Inspect the resolver output and lockfile

The project metadata and lockfile answer different questions. The former declares what the project asks for; the latter describes the resolution produced for it. In this control, inspecting the lockfile tests whether an ordinary dependency introduced any variant records. General stability and GPU behavior need their own checks.

The recorded follow-up enabled verbose output while reinstalling the same dependency:

```
uv add --preview-features wheel-variants --verbose --reinstall requests
```

Its debug output listed `wheel-variants` among the enabled preview features. That line confirms the resolver invocation enabled the feature. It doesn't identify a selected GPU artifact. In a package experiment, look separately for evidence of which distribution the resolver chose; those are two different checks.

The resulting lockfile contained no variant records. That fits the recorded dependency set, which supplied no variant metadata. The absence of those records answers the narrow lockfile question. It doesn't tell us whether the NVIDIA provider detected the card, because this dependency set didn't require a variant choice.

Choose the next experiment around a package

Suppose you're evaluating a package whose maintainer supplies NVIDIA wheel variants. This is an illustrative next decision, rather than another result from our workstation. Begin with the maintainer's exact package version, artifact source and variant requirements. Decide which selection you expect for the machine you intend to use, and write down why.

That expectation gives the experiment a purpose. If the only acceptance condition is "the install exits successfully," you can miss a successful ordinary-wheel path or a selection that doesn't exercise the intended accelerator. Decide how you'll identify the selected artifact and what small package-specific operation would demonstrate the behavior you need. Those checks depend on the package; the `requests` control doesn't supply them.

Keep a short note alongside the saved project:

```
Package version and distribution source:  
uv version and executable used:  
Interpreter and relevant host properties:  
Expected artifact and reason:  
Actual command and selected artifact:  
Package operation checked and result:
```

Use this as one experiment record, keeping the project metadata, complete lockfile and relevant command output together. If the selected artifact isn't visible, leave its field pending. To locate a failure, distinguish distribution discovery, dependency resolution, artifact installation and the package's first accelerator operation. An error at each stage asks a different question: whether the artifact was available, whether its requirements could be satisfied, whether it could be installed, or whether its code could run.

For example, a package might install successfully while its first accelerator operation fails. This is a hypothetical outcome, not another workstation result. The next check would use the package's own runtime requirements and failing operation to investigate driver compatibility or package behavior. Repeating the add command alone wouldn't exercise the operation that failed.

If your intended package has no suitable published variant metadata, the evaluation can stop after the control and source review. You've established what you could exercise and identified the missing input for the next test. Repeating ordinary installs with more familiar package names won't fill that gap.

Check the workflow edges before expanding

Two release-tagged implementation details deserve attention when choosing that next test. First, the NVIDIA variant provider calls the shared `NvidiaGpu::detect()` implementation. That shared detector chooses the highest reported compute capability among detected GPUs and carries a TODO about filtering by `CUDA_VISIBLE_DEVICES`. The comment lives in `crates/uv-torch/src/nvidia.rs`, which the provider invokes; the two source files have different responsibilities.

That matters on a machine with several GPUs. A result based on all detected cards may not describe the device your workload is meant to use. Our single-card control didn't test masking. If your evaluation depends on a particular visible-device set, check that behavior explicitly before trusting selection for it. The [shared detector at tag 0.13.6](#) gives you the exact implementation to inspect.

Second, the release-tagged compatibility tests contain an expected failure for exporting a package resolved to a wheel variant into `pylock.toml`. If that export is part of your delivery path, it belongs in your evaluation before you move toward a real environment. An installation result alone wouldn't cover the export you need afterward.

Mak read those upstream tests; we haven't run that suite here. Some NVIDIA test cases force capability values through environment variables, which lets the tests exercise specified conditions. Reading such a test provides implementation evidence, while running on your intended hardware answers an additional question. Keep those evidence types separate when explaining a result to a colleague. The [compatibility test file](#) is the source for these details.

One NVIDIA compatibility fixture makes the selection question concrete. It creates local wheels for a synthetic package named `ok`, with releases 1.0.0 and 2.0.0. Version 1.0.0 has variants labelled `gpu88` and `gpu90`; version 2.0.0 has only `gpu90`. The test forces CUDA version 12 and compute capability 8.9, then

expects the older release's `gpu88` variant. The source checks the installed fixture file for that label, so its assertion reaches beyond an installation exit code.

This fixture shows why the newest package version and the highest-looking variant label are insufficient selection rules. The declared compatibility properties constrain the choice. It also gives you a model for a package probe: identify the distribution and check something inside the installed result that distinguishes it. Those wheels are synthetic upstream test inputs, and we haven't built or installed them here. Our requests control used Python 3.11; this fixture creates its test context with Python 3.12, another reason to keep its expected result separate from the workstation observation.

When investigating a metadata error, compare the schema revision served with the package against what the evaluated uv version handles. The test files reference different revisions, so copy the exact schema identifier into your notes. That gives a maintainer a specific input to investigate alongside the package version and failing command.

| Sources & further reading

Checked 2026-10-29 · Story context 2026-10-26

1. [uv 0.13.6 release notes — preview feature: detect NVIDIA GPU capabilities when selecting compatible wheel variants](#) (2026-10-26)
2. [astral-sh/uv pull request #22690 — implement a vendored nvidia variant provider](#) (2026-10-22)
3. [astral-sh/uv 0.13.6 source — crates/uv-distribution/src/variant/nvidia.rs \(nvidia variant provider used by the wheel-variants preview\)](#) (2026-10-26)
4. [astral-sh/uv 0.13.6 tests — crates/uv/tests/pip_install/wheel_variant_compatibility.rs \(wheel-variant selection, pylock.toml export limit\)](#) (2026-10-26)
5. [uv 0.13.6 shared NVIDIA GPU detector — CUDA and compute capability queries](#) (2026-10-26)
6. [uv 0.13.6 wheel variant tests — metadata fixtures and validation](#) (2026-10-26)

[Read the current article online →](#)

Support the workshop

Explore software and learning offers at [StackSocial](#). Affiliate link: ProgreTech may earn a commission from purchases. Check current price, license and renewal terms before buying.

Find our projects at progretech.com.

ProgreTech LLC · Build / Learn / Improve / Together

This file contains its text, illustrations and fonts. No account, scripts or network connection required to read. External reference links need internet access.