Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

GPU-Accelerated 3D Non-Local Means for Python: A Toolchain-Free, Pip-Installable CuPy Implementation

Matin Mahzad

matinmahzad@yahoo.com

Independent Researcher

Article

Keywords: GPU acceleration, non-local means denoising, CuPy, Python scientific computing, real-time image processing

Posted Date: October 27th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-10296071/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License. Read Full License

Additional Declarations: No competing interests reported.

Abstract

Non-local means (NLM) denoising is implemented in scikit-image's *denoise_nl_means* function, but its reference implementation scales poorly to large 3D volumes, requiring several minutes per volume even in its optimized fast-mode configuration. GPU acceleration of 3D NLM has been demonstrated before, but only as compiled CUDA C + + tied to a multi-GPU cluster deployment or a specific imaging modality; OpenCV's CUDA module provides a GPU-accelerated 2D implementation, but the standard, pip-installable *opencv-python* ships without CUDA support, and no 3D equivalent exists there at all. No general-purpose, pip-installable Python package for GPU-accelerated 3D NLM exists today. This paper closes that gap with a single-file CuPy implementation requiring no CUDA Toolkit installation and no host C + + compiler — in contrast to PyTorch's inline-CUDA interface, which requires both. The implementation just-in-time compiles CUDA kernels through CuPy's RawKernel/NVRTC interface, a GPU-driver-only dependency, and selects, at runtime, among four shared-memory caching strategies, choosing the most aggressive tiling that fits a fixed 46 KB shared-memory budget for the requested patch parameters. On a 500×500×500 volume (NVIDIA GTX 1080 Ti), the implementation reduces wall-clock time from 420–434 s (scikit-image v0.26.0 CPU reference) to 3.25 s in float32 and 7.8 s in float64 — a 129–134× and 54–55× reduction respectively — while matching the reference output to within floating-point rounding. This moves an operation that previously forced offline, batch-only processing into a range compatible with interactive, iterative use. The software is released as open source and is a drop-in replacement for scikit-image's existing API.

1. Introduction

Non-local means (NLM) denoising, introduced by Buades, Coll, and Morel [1], estimates each voxel's value as a weighted average over other voxels whose surrounding patch is similar, rather than restricting the average to a small local neighbourhood. The method is implemented in scikit-image's *denoise_nl_means* function [2], which relies on the fast, integral-image reformulation introduced by Darbon et al. [3] and extended by Froment [4] to reduce the per-shift cost from patch-size-dependent to a small constant, independent of patch size. This fast-mode configuration is the default and recommended setting in scikit-image.

Despite this optimisation, the reference CPU implementation remains slow on 3D volumes at realistic sizes: denoising a single 500×500×500 volume with typical parameters takes 420–434 s on a modern workstation CPU (Section 5). This cost forces batch, offline workflows for any pipeline that requires 3D NLM denoising at scale and precludes interactive use.

The per-shift computation is embarrassingly parallel across voxels, making GPU acceleration a natural fit, but no GPU implementation of 3D NLM is currently reachable from a standard Python install. NVIDIA's RAPIDS/cuCIM ecosystem does not include non-local means. OpenCV's CUDA module provides a GPU-accelerated 2D non-local means function [5], but the standard, pip-installable *opencv-python* distribution ships without CUDA support at all, and source-compiled, CUDA-enabled builds have been reported missing generated Python bindings for this specific function; no 3D equivalent exists in OpenCV regardless of build configuration. Published GPU implementations of 3D NLM (Section 2) are compiled CUDA C + + tied to a specific multi-GPU deployment or a specific imaging modality, and none is distributed as a general-purpose, pip-installable Python package. Separately, CuPy [6] provides a NumPy-compatible array library with native support for both Windows and Linux — unlike RAPIDS, which targets Linux only — and exposes a RawKernel interface that compiles raw CUDA C + +

through NVRTC at runtime, avoiding any ahead-of-time build step for the end user. PyTorch [7], the other major GPU array library with native Windows and Linux support, offers no toolchain-free equivalent. Its interface for custom CUDA code, *torch.utils.cpp_extension*, compiles through nvcc, which compiles host and device code together and therefore requires a working host C + + compiler (MSVC on Windows, gcc on Linux) in addition to a CUDA Toolkit installation; an NVRTC-based alternative that would avoid this remains a private, unstable API as of this writing. The distinction is structural, not a matter of degree: CuPy's RawKernel depends on the GPU driver alone, since NVRTC compiles device code only and never invokes a host compiler, regardless of how the CUDA source is packaged.

This paper presents an implementation combining these two facts: a drop-in, single-file replacement for scikit-image's *denoise_nl_means* (3-D, fast-mode) built entirely on CuPy's RawKernel interface. The underlying mathematics is unchanged from [1–4]. The contribution is to the Python scientific-computing ecosystem specifically: a shared-memory caching strategy that adapts to the requested patch parameters at runtime, a precision pipeline that matches scikit-image's output to within floating-point rounding in both float32 and float64, and a distribution model that requires no compiled artefact to build, install, or ship. The kernel is still compiled — CuPy's RawKernel invokes NVRTC at runtime — but the unit of distribution is one human-readable .py file rather than a platform- and Python-version-specific binary, and the unit of installation is pip install rather than a source build against a local CUDA toolchain. To the authors' knowledge, no such package exists for 3D non-local means today.

2. Related Work

2.1 Fast non-local means

The original NLM formulation [1] computes, for each voxel i, a weighted average over a search window:

$$NL\left[u\right]\left(i\right)=\sum_{j\in\ \Omega}w\left(i,j\right)\ u\left(j\right),\qquad w\left(i,j\right)=\frac{1}{Z\left(i\right)}\exp\left(-\frac{\backslash\ lVertv\left(N_i\right)-v\left(N_j\right)\backslash\ rVert_{2,a}^2}{h^2}\right)$$

with computational complexity proportional to patch_size × patch_distance per voxel per dimension — prohibitive for 3D volumes at large search radii. Darbon et al. [3] reformulated the patch-distance computation using integral images (summed-area tables), reducing the per-shift cost to a small constant regardless of patch size, at the cost of restricting the patch weighting to a uniform box rather than the original Gaussian-weighted patch. Froment [4] extended this to a parameter-free pixelwise formulation, reporting a further 6–49× speedup over the original blockwise fast implementation on CPU. scikit-image's fast-mode [2] implements the integral-image approach directly and, when a noise standard deviation is supplied, applies the variance-bias correction described by Buades, Coll, and Morel [8]. This is the exact algorithm replicated in the present work; no modification to the underlying mathematics is made.

2.2 GPU acceleration of non-local means

GPU acceleration of NLM predates this work. For 3D volumes specifically, several prior efforts are directly relevant. Palma et al. [9] implemented 3D NLM across multiple GPUs in compiled CUDA C + + for a cluster deployment, and Granata et al. [10] extended that multi-GPU approach with adaptive parameter estimation for MRI Rician noise; both are reported as part of their respective studies rather than released as a redistributable,

general-purpose package, and no maintained public repository could be located for either. Li et al. [11] recently proposed a 3D NLM kernel for embedded GPUs built on nested reuse strategies across registers and global memory — a different optimisation target (resource-constrained embedded hardware) and a different technical approach (fine-grained reuse across three loop dimensions) from the shared-memory tiling cascade described in Section 4.4. Howison and Bethel [12] accelerated three 3D denoising methods — bilateral filtering, anisotropic diffusion, and non-local means — for magnetic-resonance volumes on a single GPU, with parameters tuned specifically for that imaging modality and clinical turnaround requirements. None of these four implementations is written in Python, installable without a build step, or intended as a drop-in replacement for an existing CPU library's API.

2.3 Positioning

The gap addressed here is not whether 3D NLM can run on a GPU — Section 2.2 establishes that it can — but whether it can do so without a compiled toolchain, a specific cluster or imaging-modality target, or a Linux-only dependency. CuPy's RawKernel interface [6] and NVRTC's runtime compilation make this possible: kernel source is plain text embedded in a Python module, compiled once per process on first use and cached thereafter, with identical source executing on both Windows and Linux. To the authors' knowledge, no CuPy-based implementation of non-local means, 2-D or 3-D, has previously been published.

3. Real-Time Considerations

A processing step that takes several minutes forces an offline, batch architecture: a pipeline must complete an entire denoising pass and write results to disk before a user or downstream algorithm can inspect the outcome, and any parameter change — patch size, search radius, noise estimate — requires repeating the multi-minute pass from scratch. Miller's foundational analysis of response-time thresholds in interactive systems [13], still the standard reference for this distinction, identifies roughly ten seconds as the limit past which a user's attention disengages from the task and the interaction reverts to a batch mental model; below that threshold, and particularly under about one second, the user's flow of thought continues uninterrupted. At 3–8 s per 500^3 volume, this implementation sits inside that ten-second attention threshold; the 420–434 s CPU reference sits roughly two orders of magnitude past it. That is the concrete sense in which this work changes the real-time character of the operation: a pipeline can include per-volume denoising as a step a user waits on and stays engaged with, or one that runs alongside acquisition or ingestion, rather than a separate offline stage queued for later. This is not real-time in the hard-deadline, control-systems sense; it is the specific, citable sense in which 3D NLM denoising moves from batch-only to interactive-compatible.

4. Implementation

4.1 Distribution and compilation model

The implementation is a single Python file with one runtime dependency, CuPy. The CUDA kernel source is embedded as a Python string and compiled through CuPy's RawKernel interface, which invokes NVRTC on first use within a process and caches the compiled kernel for that process's remaining lifetime. No *nvcc* invocation, build system, or platform-specific compiled binary is required; the same .py file runs unmodified on Windows and Linux, the only two platforms with native (non-WSL) CuPy wheels. Because NVRTC compiles device code only, no host C + + compiler is invoked at any point; the only system-level requirement is a CUDA-capable GPU driver.

The public function signature mirrors scikit-image's *denoise_nl_means*, including parameters (fast_mode, preserve_range, channel_axis) that this implementation accepts for drop-in compatibility without varying its behaviour.

4.2 Precision pipeline

Input is dispatched by dtype: float64 input is processed entirely in float64; any other dtype is cast to float32 and processed in float32. One step is an exception to the surrounding precision: the per-shift exponential weight uses the fast approximation of Schraudolph [14], which scikit-image also uses internally, and which requires double-precision arithmetic in its bit-manipulation step to reproduce scikit-image's output bit-for-bit. This step is therefore always computed in float64, regardless of the surrounding pipeline's precision — matching scikit-image's actual behaviour rather than a mathematically "cleaner" alternative that would not reproduce it.

4.3 Exact-output-matching engineering

Three implementation details, present in scikit-image's compiled Cython source but not documented in its public API, are replicated exactly. The candidate patch is an asymmetric $(2p)^3$ box with per-axis offset $[1-p,\ p]$, not a symmetric $(2p+1)^3$ box, a consequence of the eight-corner summed-area-table formula used for the integral-image distance computation. Normalisation divides by the nominal patch volume s^3 (the requested, always-odd patch_size) even though the summed box contains only $(2p)^3$ voxels, while the optional noise-variance correction of [7] subtracts $2\sigma^2$ scaled by the actual box volume $(2p)^3$ — two different volumes in the same normalisation step. The exponential weight is computed via the Schraudolph approximation [12] rather than a standard exponential call. Matching these choices, rather than a more mathematically direct re-implementation, is what makes the GPU output numerically indistinguishable from the CPU reference (Section 5) rather than merely visually similar.

4.4 Shared-memory caching cascade

The kernel loads a padded candidate tile into shared memory once per thread block and reuses it across the inner search loop rather than re-reading global memory for every candidate shift. How much of the search-and-patch neighbourhood can be cached this way is bounded by a fixed shared-memory budget:

$$T_{ex}^3 \cdot \text{elem_bytes} \ \le\ 46 \times 1024 \text{ bytes}, \qquad T_{ex} = b + 2(p+d)$$

leaving headroom under the 48 KB default per-block shared-memory limit on Pascal-generation GPUs, including the GTX 1080 Ti used for testing. This budget is a function of block size b, patch_size, patch_distance, and element width (4 or 8 bytes), so the implementation selects, at runtime, the most aggressive of four caching tiers whose footprint fits the requested parameters at some valid block size $b^3 \le 1024$:

Tier 3 caches the entire search-plus-patch neighbourhood in one tile, loaded once per block with no reload inside the search loop; its footprint scales with T_{ex}^3 and it is the fastest tier where it fits. Tiers 2 and 1 extend the candidate tile along two or one axes respectively, trading reduced caching for a smaller footprint. Tier 0 reloads the full tile on every shift and has the smallest footprint of the four, serving as a universal fallback. Block sizes are searched from 10 down to 1 for each tier in turn, most aggressive first, and the first combination that fits is used — guaranteeing that any parameter combination valid under the smallest-footprint tier remains runnable,

while opportunistically using a faster tier whenever the budget allows. For the benchmark parameters used in Section 5 (patch_size = 3, patch_distance = 3), Tier 3 is selected on the test GPU. Figure 1 illustrates the tier-selection logic and the resulting shared-memory tile layout for each tier.

Two implementation observations shaped this design. First, moving the inner-loop accumulation from float64 to float32 alone produced only a 1.8× speedup, far below the 32× difference in Pascal's FP64:FP32 throughput, indicating the kernel is memory-bound rather than compute-bound at these parameters. Second, in an earlier revision, caching the full candidate tile once per block measured no faster than reloading it on every shift, despite an 84× difference in raw elements read per block — consistent with the L2 cache already absorbing most redundant traffic at small search radii. Both observations motivated targeting memory-access pattern rather than arithmetic precision, which the tiered cascade addresses.

5. Results

All measurements use a synthetic 500×500×500 volume, patch_size = 3, patch_distance = 3, h = 0.1, sigma = 0.0, on an NVIDIA GTX 1080 Ti (Pascal, Compute Capability 6.1). The CPU reference is scikit-image v0.26.0's *denoise_nl_means* with fast_mode=True, run single-threaded as shipped. Each session times three consecutive calls, of which the first includes one-time NVRTC compilation (Fig. 2a); reported GPU times are steady-state means of the second and third calls. GPU steady-state timing on this hardware showed run-to-run variation of up to ≈ 20% between separate process invocations, plausibly attributable to clock-boost ramp state or thermal conditions rather than to the code, so results are confirmed across two independent process invocations rather than one (Table 1, Fig. 2b).

Table 1
Runtime and accuracy, 500³ volume, two independent process invocations.

Path	Time (s)	Speedup vs. CPU	Max abs. diff vs. reference
CPU — scikit-image v0.26.0 (reference)	420–434	1×	—
GPU float32 (Tier 3)	3.25	129–134×	5.96×10^{-8}
GPU float64 (Tier 3)	7.82–7.83	54–55×	3.21×10^{-8}

The GPU float32 path completes in 3.25 s against a 420–434 s CPU reference (129–134×); the float64 path completes in 7.82–7.83 s (54–55×). The float64:float32 GPU time ratio (2.41×) is far below the 32× difference in Pascal's raw FP64:FP32 throughput, consistent with the memory-bound behaviour discussed in Section 4.4.

Output correctness was evaluated as the maximum absolute voxel-wise difference against the scikit-image reference: 5.96×10^{-8} (float32) and 3.21×10^{-8} (float64). Both figures are consistent with, and slightly below, the 6.7×10^{-8} upper bound established for Tier 3 specifically during per-tier kernel validation, and reflect floating-point accumulation-order differences introduced by Tier 3's more aggressive caching rather than any algorithmic deviation: Tiers 0–2, which do not restructure accumulation order, matched the CPU reference exactly or to double-precision machine epsilon in isolated per-tier testing. All speedups reported above use Tier 3, which is selected automatically for the benchmark's patch_size/patch_distance combination.

6. Discussion

This implementation targets a narrower goal than either prior GPU effort discussed in Section 2.2: not multi-GPU scaling, and not tuning for a specific imaging modality, but single-GPU acceleration distributed as an ordinary, pip-installable Python package with no compiled artefact to build or ship. This trades some ceiling — a hand-tuned, modality-specific compiled kernel could plausibly go faster still, and multi-GPU scaling is out of scope here — for accessibility: a user with scikit-image and a CUDA-capable GPU on Windows or Linux can install and use this implementation without a compiler toolchain, without WSL, and without changing existing code beyond the import statement.

The current implementation targets a single GPU and the fast-mode formulation only; the original, non-integral-image algorithm (fast_mode=False in scikit-image) is not implemented, since its computational profile and memory-access pattern differ substantially and would warrant a separate design. Tier selection was validated on Pascal-generation hardware (46 KB usable shared memory per block); GPUs with a larger shared-memory budget would be expected to select Tier 3 for a wider range of patch_size/patch_distance combinations without code changes, though this has not been tested on other architectures.

7. Conclusion

This work closes a specific, verifiable gap: a general-purpose, toolchain-free GPU implementation of 3D non-local means denoising was not previously available to the Python ecosystem as redistributable, pip-installable software. The implementation matches scikit-image's reference output to within floating-point rounding while reducing wall-clock time on a 500^3 volume from over seven minutes to under eight seconds, moving the operation from batch-only to interactive-compatible, with no CUDA Toolkit installation or host compiler required beyond the GPU driver. The software is released as open source and is available at the repository below.

Declarations

Funding

No funding was received to assist with the preparation of this manuscript.

Conflict of interest

The authors declare no competing financial or non-financial interests.

Code and Data Availability

Name of the code/library: nlm3d_cuda -- GPU-Accelerated 3D Non-Local Means Denoising

Contact: matinmahzad@yahoo.com

Hardware requirements: Standard CPU with an NVIDIA CUDA-compatible GPU

Programming language: Python

Software required: CuPy, NumPy

Program size: ~37 KB

The source code and data are available at: https://github.com/SeismicFlow/nlm3d-cuda

Author Contribution

M.M. conceived and designed the study, developed the method and software, performed the analysis, and wrote the manuscript.

Data Availability

Code and Data AvailabilityName of the code/library: nlm3d_cuda -- GPU-Accelerated 3D Non-Local Means DenoisingContact: matinmahzad@yahoo.comHardware requirements: Standard CPU with an NVIDIA CUDA-compatible GPUProgramming language: PythonSoftware required: CuPy, NumPyProgram size: ~37 KBThe source code and data are available at: https://github.com/SeismicFlow/nlm3d-cuda

References

1. Buades, A. & Coll, B. J.-M. Morel. A non-local algorithm for image denoising. In: 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05), Vol. 2, pp. 60–65 (2005). 10.1109/CVPR.2005.38
2. van der Walt, S. et al. the scikit-image contributors. scikit-image: image processing in Python. PeerJ 2:e453 (2014). 10.7717/peerj.453
3. Darbon, J., Cunha, A., Chan, T. F., Osher, S. & Jensen, G. J. Fast nonlocal filtering applied to electron cryomicroscopy. In: 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro (ISBI'08), pp. 1331–1334 (2008). 10.1109/ISBI.2008.4541250
4. Froment, J. Parameter-Free Fast Pixelwise Non-Local Means Denoising. *Image Process. Line* **4**, 300–326 doi:10.5201/ipol.2014.120 (2014).
5. Bradski, G. The OpenCV Library. Dr. Dobb's Journal of Software Tools (2000).
6. Okuta, R., Unno, Y., Nishino, D., Hido, S. & Loomis, C. CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations. Proceedings of the Workshop on Machine Learning Systems (LearningSys) at NeurIPS (2017). (2017).
7. Paszke, A. et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Adv. Neural Inform. Process. Syst. (NeurIPS)*. **32**, 8024–8035 (2019).
8. Buades, A. & Coll, B. Morel. Non-Local Means Denoising. *Image Process. Line*. **1**, 208–212. 10.5201/ipol.2011.bcm_nlm (2011).
9. Palma, G. et al. P. Borrelli. 3D Non-Local Means denoising via multi-GPU. In: Proceedings of the 2013 Federated Conference on Computer Science and Information Systems (FedCSIS), pp. 495–498 (2013).
10. Granata, D., Amato, U. & Alfano, B. MRI denoising by nonlocal means on multi-GPU. *J. Real-Time Image Proc.* **16**, 523–533. 10.1007/s11554-016-0566-2 (2019).

11. Li, X., Chang, Q. & Li, Y. J. Miyazaki. 3D GNLM: Efficient 3D Non-Local Means Kernel with Nested Reuse Strategies for Embedded GPUs. ACM Transactions on Architecture and Code Optimization 22(3), Article 90 (2025). 10.1145/3744909
12. Howison, M. & Bethel, E. W. GPU-accelerated denoising of 3D magnetic resonance images. *J. Real-Time Image Proc.* **13**, 713–724. 10.1007/s11554-014-0436-8 (2014).
13. Miller, R. B. Response time in man-computer conversational transactions. Proceedings of the AFIPS Fall Joint Computer Conference 33, 267–277 (1968). 10.1145/1476589.1476628
14. Schraudolph, N. N. & Fast, A. Compact Approximation of the Exponential Function. *Neural Comput.* **11** (4), 853–862. 10.1162/089976699300016467 (1999).

Figures

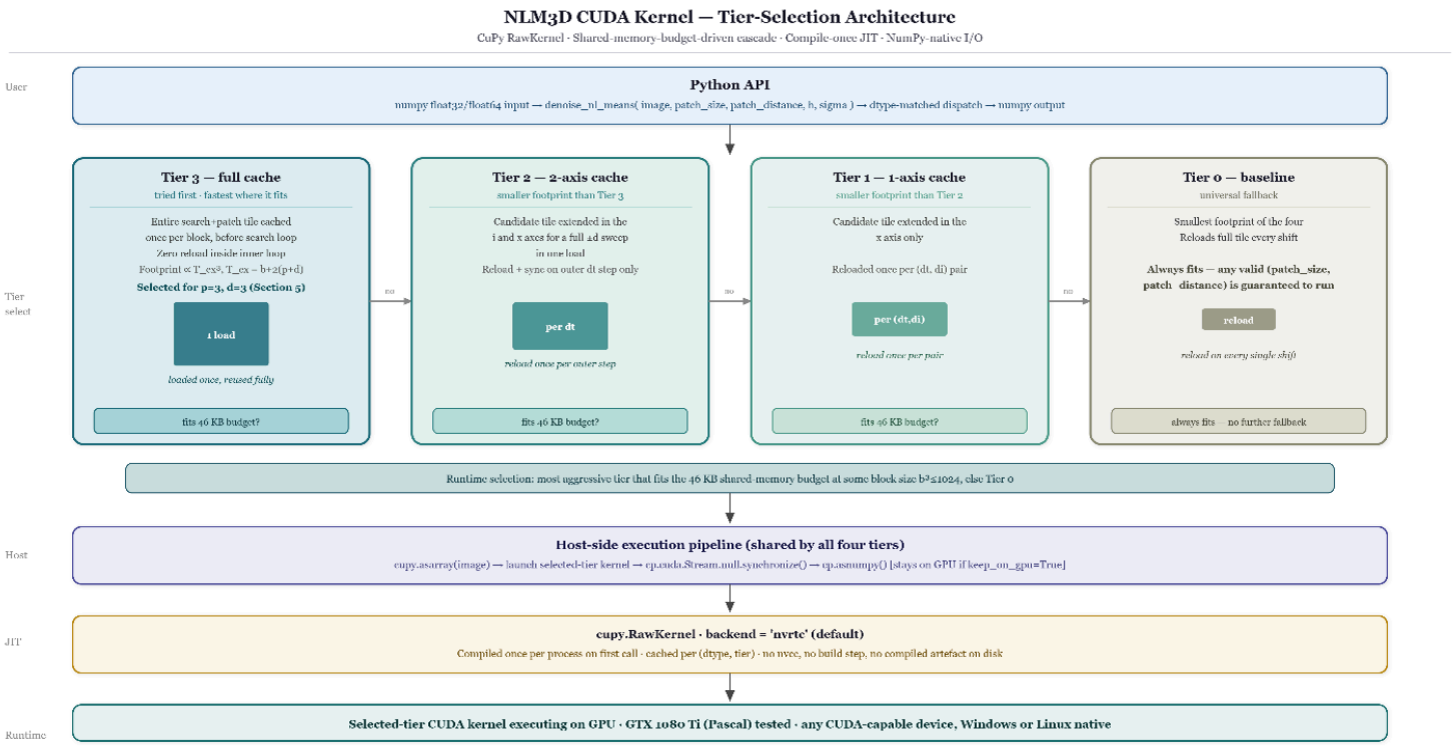

Figure 1

Tier-selection decision flow and shared-memory tile layout, Tiers 0–3.

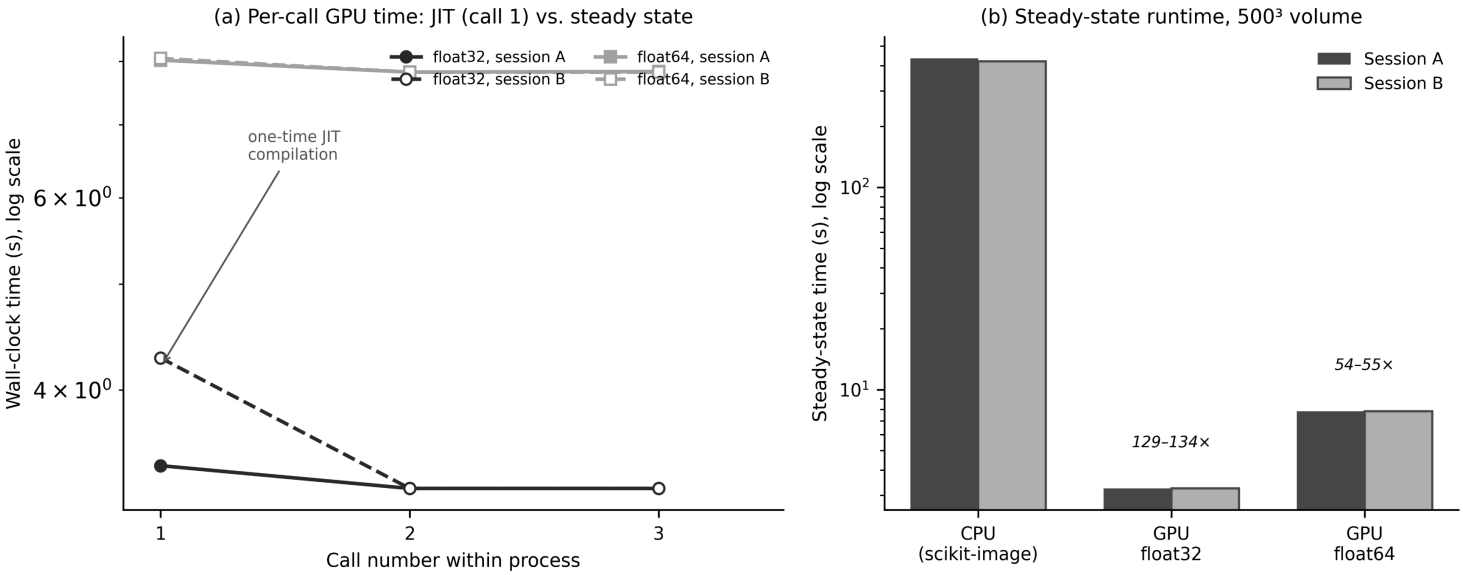

Figure 2

(a) Per-call GPU time across both independent sessions, showing the one-time JIT-compilation cost on the first call and flat steady-state timing thereafter. (b) Steady-state runtime versus the scikit-image CPU reference, both sessions.

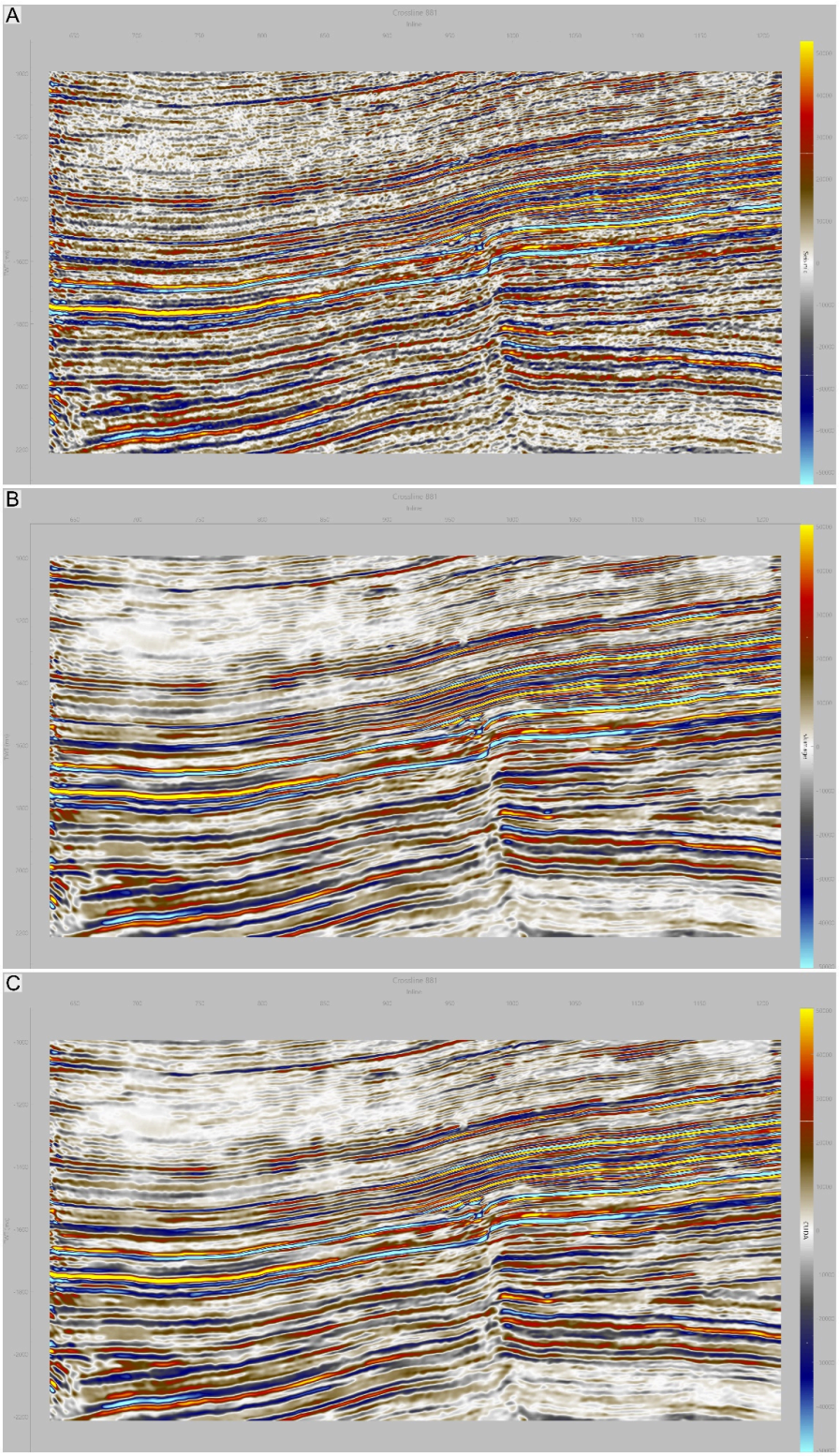

Figure 3

Qualitative correctness check on real 3D seismic data, comparing scikit-image (CPU) and GPU implementations for a representative slice. (A) Original noisy input slice. (B) Output from the scikit-image (CPU) implementation. (C) Output from the GPU (CUDA) implementation. The two outputs are visually indistinguishable, consistent with the floating-point-level differences reported in Table 1. Visualizations were generated using the SeismicFlow application.