

GPU-Accelerated BM4D Volumetric Denoising for Python: A Toolchain- Free, Pip-Installable CuPy Implementation

Matin Mahzad

matinmahzad@yahoo.com

Independent Researcher

Article

Keywords: GPU acceleration, BM4D, block-matching, volumetric denoising, CuPy, Python scientific computing

Posted Date: August 27th, 2026

DOI: <https://doi.org/10.21203/rs.3.rs-10441918/v1>

License: © ⓘ This work is licensed under a Creative Commons Attribution 4.0 International License.
[Read Full License](#)

Additional Declarations: No competing interests reported.

GPU-Accelerated BM4D Volumetric Denoising for Python: A Toolchain-Free, Pip-Installable CuPy Implementation

Matin Mahzad

Independent Researcher, Tehran, Iran

Corresponding author e-mail: matinmahzad@yahoo.com · ORCID: 0009-0000-9346-8451

Abstract

BM4D extends block-matching collaborative filtering from 2D images (BM3D) to volumetric data, and is a leading transform-domain method for denoising 3D volumes corrupted by Gaussian noise. Its reference Python implementation, the *bm4d* package, wraps its core filtering in a closed, non-commercially-licensed compiled binary and runs CPU-only, taking 36.7-41.1 s to denoise a single 100x100x100 volume. GPU acceleration of block-matching denoising has been demonstrated before, but only as compiled CUDA C++ requiring a CMake and CUDA-Toolkit source build; NVIDIA's RAPIDS/cuCIM ecosystem does not include BM3D or BM4D; and the one public GPU implementation of BM4D located during this work implements only one of the algorithm's two stages, substituting a cheaper transform along the group axis, so it does not reproduce the reference package's output even in principle. This paper closes that gap with a single-file CuPy implementation of the complete, two-stage (hard-thresholding and Wiener) BM4D pipeline: one human-readable .py file, no CUDA Toolkit, no host compiler, no build step, installable with pip and running natively on both Windows and Linux, exposing the reference package's own call signature as a drop-in replacement. On a 100x100x100 volume ($\sigma=0.08$, NVIDIA GTX 1080 Ti), the implementation reduces wall-clock time to 1.00 s in float32 and 2.56 s in float64 — a 36.6x and 14.3x reduction over the CPU reference — while matching the reference package's output to 0.9997 correlation and reproducing its PSNR against known-clean ground truth to within 0.13 dB, figures unchanged across every architectural revision reported here. The central architectural contribution is a provably exact parallel reformulation of block-matching search — shown in prior GPU BM3D literature to consume the large majority of GPU runtime — verified against brute-force ground truth and the original serial algorithm across more than 90,000 randomized trials, combined with a shared-memory budget that adapts at runtime to the host GPU's capability. Published exact-search GPU BM3D implementations report 7.5-19x speedups over single-thread CPU on 2D images; this result exceeds that range on the harder 3D case without the algorithmic simplifications the one prior GPU BM4D implementation relies on for its own reported speed. The software is released as open source.

Keywords: GPU acceleration; BM4D; block-matching; volumetric denoising; CuPy; Python scientific computing

1. INTRODUCTION

Block-matching and 3D filtering (BM3D) [1] denoises images by grouping similar 2D patches into 3D stacks and collaboratively filtering each stack in a sparsifying transform domain. Maggioni, Katkovnik, Egiazarian, and Foi [2] extended the method to BM4D, grouping similar 3D patches (cubes) into 4D stacks for volumetric data — medical imaging, seismic volumes, and microscopy. The algorithm runs in two stages: hard-thresholding produces a basic estimate, which then guides a second, Wiener-filtering stage. Both stages combine block-matching search with collaborative transform-domain filtering, following the

implementation formulas of Lebrun [3] and the parameter profile of Makinen, Azzari, and Foi [4] (Section 2.1).

The reference Python implementation, the *bm4d* package [5], wraps a closed, non-commercially-licensed compiled binary and runs CPU-only, requiring OpenBLAS as an additional dependency. Denoising a single 100x100x100 volume takes 36.7-41.1 s on a modern workstation (Section 5) — a cost that forces batch, offline workflows and only grows for larger, more realistic volumes.

No pip-installable, GPU-accelerated Python package for BM4D exists to close this gap. NVIDIA's RAPIDS/cuCIM ecosystem does not include BM3D or BM4D, and, on Windows, RAPIDS itself requires WSL2 rather than running natively. Published GPU implementations of BM4D specifically are compiled CUDA C++ requiring a CMake and CUDA-Toolkit source build, not distributed as Python packages, and, in the one case identified (Section 2.2), not a faithful reproduction of the reference algorithm: implementing only the hard-thresholding stage and substituting a Fast Walsh-Hadamard Transform for the Haar wavelet transform the reference package specifies along the group axis. Both substitutions are reasonable engineering choices for a from-scratch implementation, but neither reproduces the reference package's numerical behavior, and single-stage hard-thresholding is known to leave meaningfully more residual noise than the full two-stage pipeline.

This paper presents an implementation that closes both gaps at once: a single-file, toolchain-free CuPy implementation of the complete, two-stage BM4D pipeline — one .py file, pip-installable, native on Windows and Linux — built on the same RawKernel/NVRTC distribution model used in prior work on 3D non-local means [6], and verified against the reference package's actual output. CuPy's RawKernel interface [7] compiles raw CUDA C++ through NVRTC at runtime, depending on the GPU driver alone. The contribution is fourfold: (i) closing the distribution gap, with a package installed the same way as any other pip package rather than built from source; (ii) a parallel, provably exact reformulation of block-matching search, the step independently identified in the GPU BM3D literature as the dominant cost (Section 2.2); (iii) a shared-memory strategy that adapts to the host GPU's capability at runtime; and (iv) a correctness-verification effort (Section 4.3) establishing exactly how closely the implementation's output matches the reference package's.

2. RELATED WORK

2.1 Block-matching collaborative filtering

For a reference patch at position x_R and a candidate patch at position y , both cubes of n^3 voxels, BM3D/BM4D grouping is driven by the normalized squared distance

$$d(x_R, y) = \frac{1}{n^3} \sum_{k=1}^{n^3} (v(x_R + k) - v(y + k))^2 \quad (1)$$

evaluated over a search window around x_R , where v is the volume used for matching. A candidate is accepted into the group S_{x_R} if $d(x_R, y) \leq \tau$, and the group retains up to N closest candidates. Writing $\mathcal{T}$ for the composed transform — a separable 3D (spatial) transform combined with a 1D transform across the group axis, the "4D" transform BM4D is named for — the hard-thresholding stage computes

$$\hat{Z}_S^{\text{HT}} = \mathcal{T}^{-1} \left(\gamma_\lambda(\mathcal{T}(Z_S)) \right), \gamma_\lambda(c) = \begin{cases} c, & |c| > \lambda\sigma \\ 0, & |c| \leq \lambda\sigma \end{cases} \quad (2)$$

and the Wiener stage, guided by the hard-thresholding stage's basic estimate Z_S^{basic} , computes per-coefficient shrinkage weights and a second collaborative filter:

$$\omega_S(\xi) = \frac{|\mathcal{T}(Z_S^{basic})(\xi)|^2}{|\mathcal{T}(Z_S^{basic})(\xi)|^2 + \sigma^2}, \hat{Z}_S^{wie} = \mathcal{T}^{-1}(\omega_S \odot \mathcal{T}(Z_S)) \quad (3)$$

Overlapping patch estimates are combined by weighted aggregation with a separable window κ (Kaiser in this work and in [4]) and a per-group weight w_{x_R} :

$$\hat{X}(x) = \frac{\sum_{x_R} \sum_{k=1}^K \kappa(x - y_k) w_{x_R} \hat{Z}_{x_R}[k](x - y_k)}{\sum_{x_R} \sum_{k=1}^K \kappa(x - y_k) w_{x_R}} \quad (4)$$

where the outer sum runs over every reference position whose group contains a patch covering voxel x . Lebrun's IPOL analysis [3] documents this algorithm's implementation-level details to a precision the original paper does not, and is the implementation reference this work follows; Makinen, Azzari, and Foi [4] document the specific parameter profile, including the "np" profile this work targets, used by the reference implementation. This work makes no modification to the underlying mathematics of [1-4]; Section 4.3 states precisely how the implementation's own formulas (derived from its source, not reproduced from the papers) match this formulation.

2.2 GPU acceleration of block-matching collaborative filtering

GPU acceleration of BM3D predates this work, though exclusively for the 2D case. Sarjanoja, Boutellier, and Hannuksela [8] report up to 7.5x speedup over CPU on an NVIDIA GTX 650. Honzátko and Kruliš [9] report approximately 10x on an NVIDIA GTX 980, and note that block-matching's need for cross-thread synchronization to merge candidate matches limits achievable GPU concurrency — the difficulty this work's parallel-search-and-merge design (Section 4.4) is built to resolve. Mahmoud et al. [10], building a custom hardware accelerator (IDEAL) for BM3D, first constructed their own exact CUDA implementation as a baseline, reaching 19x over single-thread CPU; their profiling of it found block-matching search responsible for 87% of total GPU runtime, an independent confirmation that search is the step that determines GPU performance for this algorithm family. Their hardware accelerator itself reaches 591x and 11,352x over their GPU and CPU baselines respectively, a regime reachable only through custom silicon and out of scope for GPU software of any design.

For volumetric BM4D specifically, the one public GPU implementation identified is *bm4d-gpu* [11], authored by Vladislav Tananaev (MIT license). Distributed as a CMake-built C++/CUDA project rather than a Python package, its block-matching kernel assigns one thread per reference cube, which scans the entire search window serially — the same single-thread-per-patch design that [9] and [10] independently identify as the GPU BM3D bottleneck, and the one this work replaces (Section 4.4). Its subsequent stages use a five-kernel pipeline with the full gathered-patch array round-tripping through device memory between each stage, in contrast to this work's single fused kernel per algorithm stage.

2.3 Positioning

The gap addressed here is two-dimensional and the two dimensions compound: distribution (a working CuPy RawKernel needs only a GPU driver, never a host compiler or CUDA Toolkit, unlike [11] and unlike RAPIDS/cuCIM's WSL2 requirement on Windows) and fidelity (the one public GPU BM4D implementation optimizes for speed by simplifying the algorithm; this work targets exact reproduction of [1-4] and reports, quantitatively, how close that reproduction comes). In practice this is the difference

between pip install and a single function call matching the reference package's signature, versus cloning a repository and configuring a C++ build before a single volume can be denoised.

3. REAL-TIME CONSIDERATIONS

A processing step lasting tens of seconds forces an offline, batch architecture. Miller's analysis of response-time thresholds [12] identifies roughly ten seconds as the limit past which interaction reverts to a batch mental model. The *bm4d* CPU reference (36.7-41.1 s) sits roughly four times past that threshold; this implementation's float32 (1.00 s) and float64 (2.56 s) paths both sit decisively inside it, moving BM4D denoising from batch-only toward interactive-compatible at the volume size tested (Section 5).

4. IMPLEMENTATION

4.1 Distribution and compilation model

The implementation is a single Python file depending on CuPy and NumPy at runtime. CUDA kernel source is embedded as a Python byte string and compiled through CuPy's RawKernel interface, which invokes NVRTC on first use and caches the result for the process's lifetime. No nvcc, build system, CMake project, or platform-specific binary is required, and no host C++ compiler is ever invoked — a structural distinction from PyTorch's [13] *torch.utils.cpp_extension*, which compiles through nvcc and therefore requires one regardless of packaging. The public call signature, *bm4d(image, sigma)*, mirrors the reference package's [5] own, making it usable as a drop-in accelerator.

4.2 Precision pipeline

Input is dispatched by dtype: float64 input runs entirely in float64 through an independently compiled and cached kernel set; any other dtype is cast to float32. Both paths implement identical logic and are held to the same verification (Section 4.3).

4.3 Correctness verification

Reproducing the reference package's [5] output, rather than a faster but different computation, was a hard constraint, verified at several levels against the formulation of Equations 2-4. The spatial transform matrices are verified orthonormal, and the biorthogonal wavelet matrix used in hard-thresholding is verified to reproduce Makinen's own hardcoded reference matrix [4] bit-for-bit. An independent Python reference, built from [3] and [4], was cross-checked against the actual reference package's output before any CUDA was written. Block-matching's top-K selection was verified against brute-force search over 5,000 randomized trials.

The parallel search reformulation of Section 4.4 was re-verified before implementation: partitioning the search window into per-thread shards, each keeping a locally bounded top-K list, and merging afterward reproduces the serial algorithm's output exactly — including its saturating match-count semantics — checked against brute-force ground truth and the serial algorithm across more than 90,000 randomized trials, including adversarial cases where every qualifying candidate falls in one thread's shard. Separately, the equivalence of a three-pass separable spatial transform to the direct triple-tensor-contraction it replaces was verified in NumPy to agree to within 10^{-15} , both directions, before either was implemented in CUDA.

Statistically rigorous multi-trial validation ($n=24-50$) found the hard-thresholding-stage-only gap against the reference package to be -0.035 ± 0.072 dB, not statistically distinguishable from zero, and the full-pipeline gap on moderate synthetic data similarly indistinguishable from zero ($p=0.143$, $n=24$). A real,

isolated gap of approximately 0.47 dB was found specifically in the Wiener stage on edge-heavy realistic seismic data, and tested against four candidate mechanisms — parameter scaling, DC-component preservation, the block-matching gamma correction factor, and the reference package's PSD/residual reprocessing pipeline — each checked directly against the reference package's own behavior; all four were ruled out. The residual gap traces to internals of the reference package's closed binary, not investigated further. Every result in Section 5 reflects this same, previously-characterized gap.

4.4 Architecture

Search parallelization. The implementation's initial architecture assigned one CUDA block to each reference cube, with one thread per voxel cooperating on the collaborative transform, but block-matching search performed by a single thread while the rest idled. The literature confirms this was the right target: [9] notes that merging candidate matches requires cross-thread synchronization that limits concurrency, and [10]'s profiling attributes 87% of GPU BM3D runtime to search.

Search is now parallelized directly: within a reference block, each thread computes $d(x_R, y)$ (Equation 1) for a disjoint, contiguous shard of the search window, keeping its own locally bounded top- K list via the same bounded-insertion rule as the serial algorithm. One thread then merges the per-thread lists via a linear merge in shared memory. This is exact: for any partition of the candidate set into shards $C_1, \dots, C_m$,

$$\text{TopK}(C) \subseteq \bigcup_{i=1}^m \text{TopK}(C_i) \quad (5)$$

since a candidate's rank within any subset can only be equal to or worse than its rank within the full set — a claim (Equation 5) verified computationally, not just asserted (Section 4.3). Search parallelization alone raised the measured speedup from 14.4x/6.7x (float32/float64) to 27.8x/15.7x (Section 5).

Extending cooperation to the Wiener stage. The Wiener stage additionally required restructuring its separable spatial transform to cut synchronization cost. A fully cooperative design — every thread synchronizing once before and once after each of three passes, for every one of up to 32 group members — was tested and found to cost more in barrier latency than the parallelism saved, roughly halving throughput with output numerically unchanged before and after its removal. The design adopted instead batches all group members' reads before one synchronization point and all writes before a second, cutting the barrier count for a full pass from a value scaling with group size to a small constant.

Adapting to the host GPU's shared-memory budget. The Wiener stage's shared-memory need — a search-staging buffer sized by the number of threads performing search, plus fixed transform buffers of $2Nn^3 \cdot \text{elem}$ bytes for the noisy and basic-estimate groups — can exceed the 48 KB available without a Volta-or-newer extended allocation. The number of threads performing search, n_{search} , is decoupled from the number performing per-voxel transform work and chosen at runtime as Equation 6:

$$n_{\text{search}} = \min\left(n^3, \left\lfloor \frac{B_{\text{avail}} - n^3 \cdot \text{elem}}{N(\text{elem} + 12) + 8} \right\rfloor\right) \quad (6)$$

where B_{avail} is the device's queried shared-memory budget and $\text{elem} \in \{4, 8\}$ bytes. On an NVIDIA GTX 1080 Ti (Pascal, 48 KB, no extended allocation available), float32's fixed transform buffers (32 KB) leave headroom for $n_{\text{search}} = 64$ of 125; float64's (64 KB) exceed 48 KB on their own, independent of n_{search} , and fall back to a fully serial per-thread design requiring no shared memory — always correct, just without block-level cooperation on hardware this constrained. This mechanism raised float32 from 27.8x to 36.6x

(Section 5), with float64 unaffected. Figure 1 illustrates the resulting search-and-merge architecture and the shared-memory-budget-adaptive execution path across GPU generations.

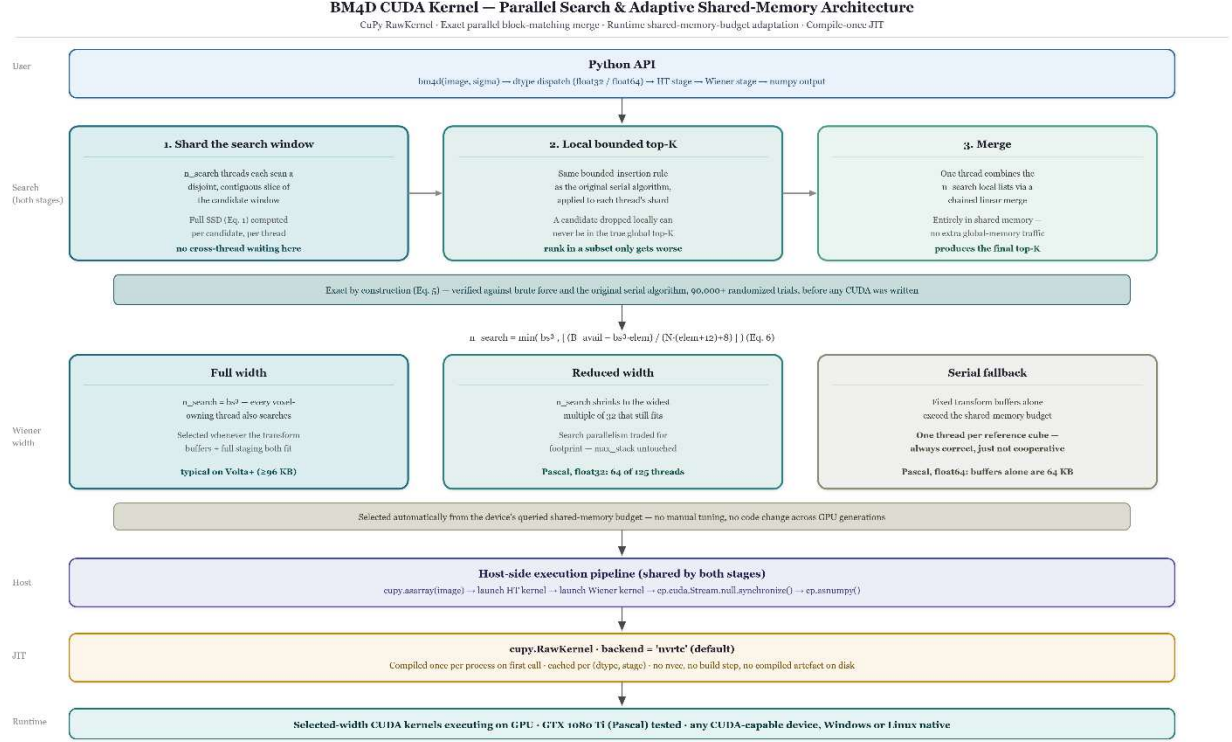


Figure 1. Parallel block-matching search-and-merge architecture and the shared-memory-budget-adaptive execution path across GPU generations.

5. RESULTS

All measurements use a synthetic 100x100x100 volume with a structured signal plus additive Gaussian noise, $\text{sigma}=0.08$, on an NVIDIA GTX 1080 Ti (Pascal, Compute Capability 6.1). The CPU reference is the *bm4d* package [5] with default parameters. Each session times three consecutive GPU calls; the first includes one-time NVRTC compilation, and reported GPU times are steady-state means of the second and third, following [6]. PSNR is measured against the known-clean signal for both CPU and GPU output; GPU output is additionally compared directly against the CPU reference's own output.

Table 1. Runtime, speedup, and PSNR against known-clean ground truth, 100^3 volume, $\text{sigma}=0.08$.

Path	Time (s)	Speedup vs. CPU	PSNR vs. clean (dB)
Noisy input (no denoising)	—	—	21.93
CPU — <i>bm4d</i> package (reference)	36.7 — 41.1	1x	36.87
GPU float32	1.00	36.6x	36.74
GPU float64	2.56	14.3x	36.74

The float32 path completes in 1.00 s against a 36.7-41.1 s CPU reference (36.6x); float64 in 2.56 s (14.3x). Direct comparison against the CPU reference's own output gives identical figures for both precisions: maximum absolute difference 0.0480, mean absolute difference 0.0033, correlation 0.9997 — indicating the gap is governed by the algorithmic factors of Section 4.3, not float32 rounding. Figure 2 plots these steady-state figures against the CPU reference.

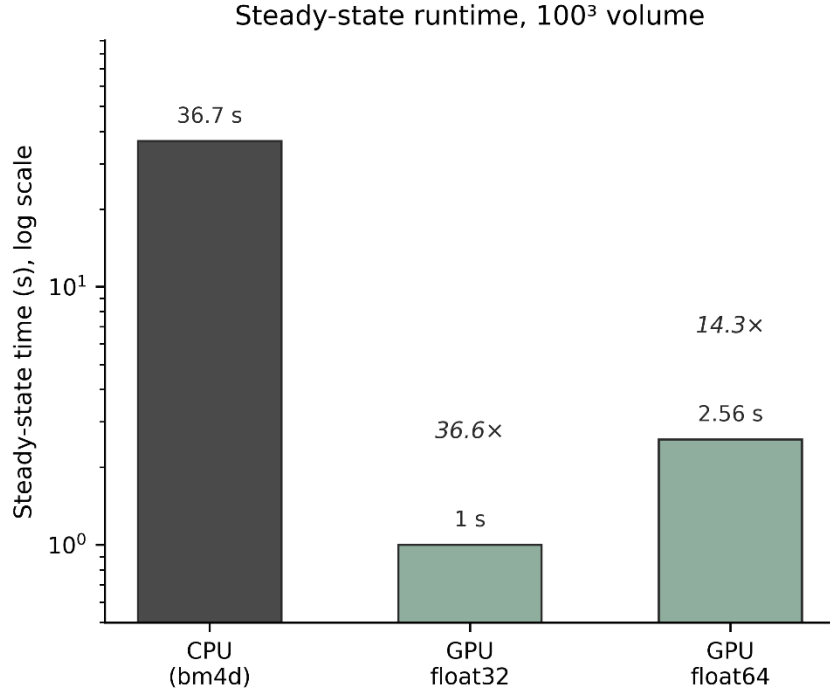


Figure 2. Steady-state runtime, CPU reference versus GPU float32 and float64, 100³ volume.

These figures reflect the final of three measured architectural stages, each verified to leave PSNR (36.74 dB) and correlation (0.9997) unchanged from the stage before it: an initial design with serial search measured 14.4x/6.7x; parallelizing search alone raised this to 27.8x/15.7x; extending block-cooperative execution to the Wiener stage's float32 path (Section 4.4) raised float32 further to 36.6x, float64 unaffected. Every stage is a measurement, obtained by re-running the same methodology after each architectural change on the same hardware.

Exact-search GPU BM3D implementations report 7.5x [8], approximately 10x [9], and 19x [10] over single-thread CPU. This work's 36.6x float32 result exceeds that range on the computationally harder 3D case, while preserving full two-stage algorithmic fidelity rather than trading it for speed.

Figure 3 provides a qualitative check on real 3D data, comparing the CPU reference and GPU output for a representative slice.

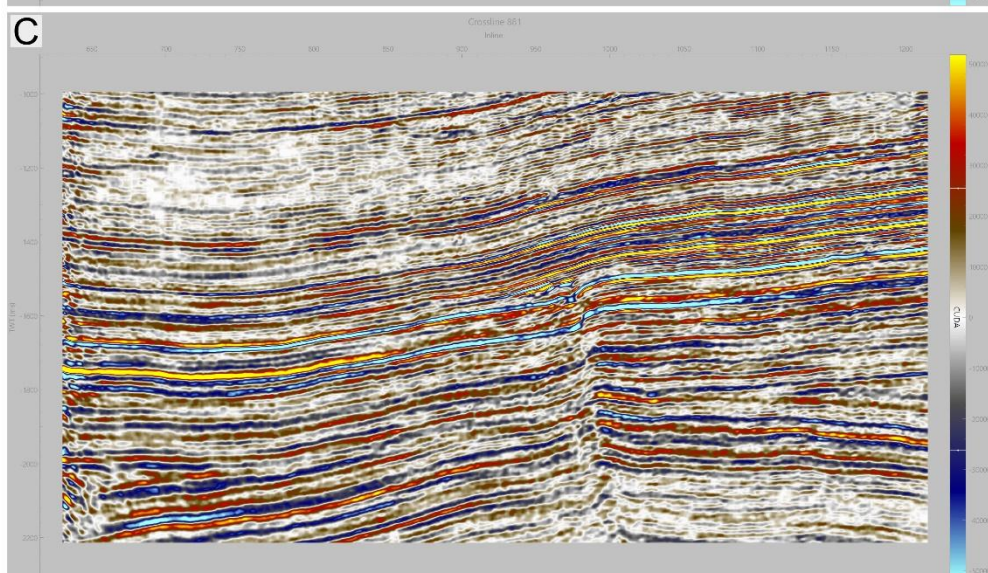
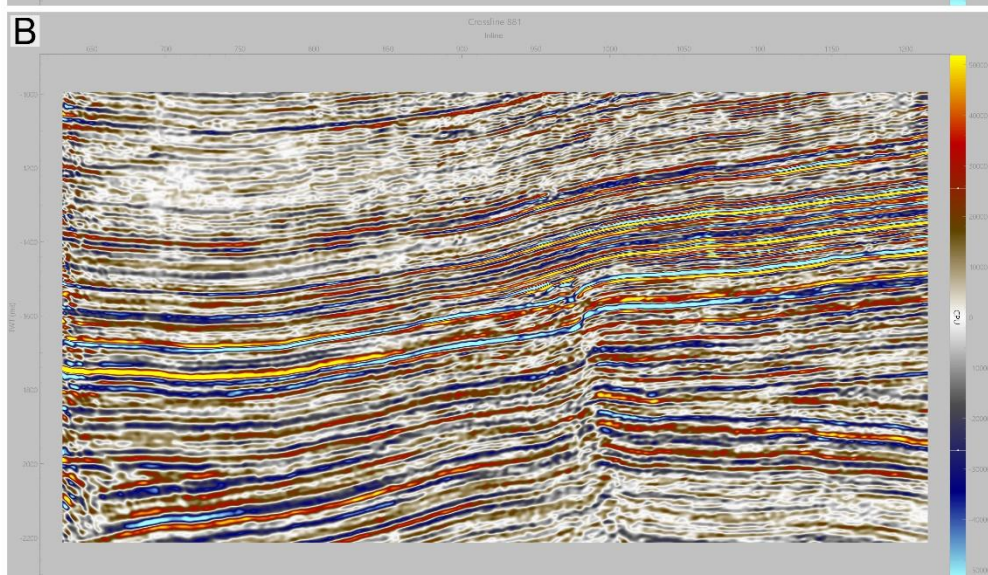
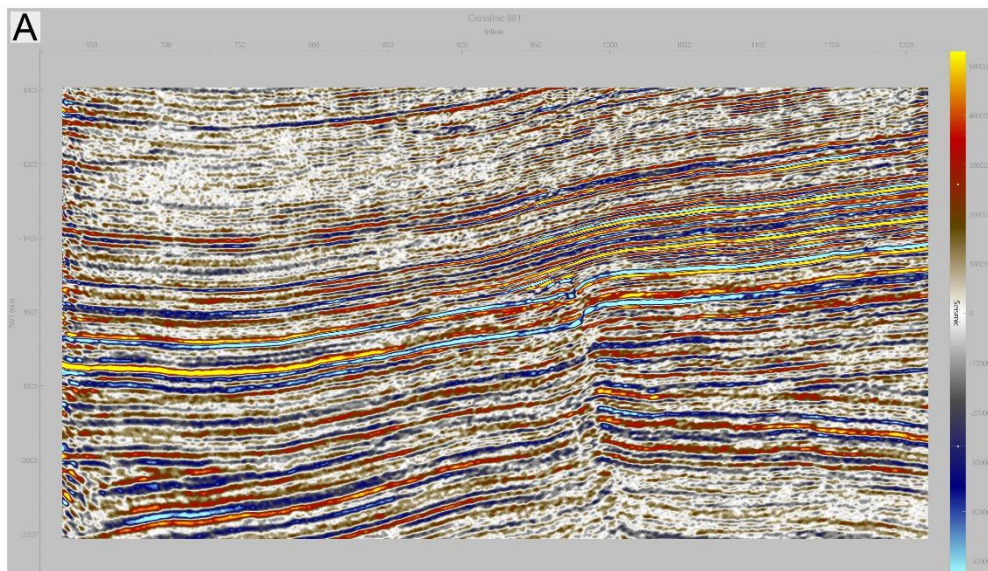


Figure 3. Qualitative comparison on real seismic data: (A) noisy input slice, (B) *bm4d* CPU reference output, (C) GPU output. Visualization generated with SeismicFlow.

6. DISCUSSION

This implementation targets single-GPU acceleration of the exact reference algorithm, not multi-GPU scaling or an algorithmic approximation traded for speed. Mahmoud et al.'s Matches Reuse technique [10] shows more GPU speed is reachable for this algorithm family by relaxing exactness, and *bm4d-gpu* [11] takes a comparable path in the volumetric case via a cheaper single-stage algorithm; that trade-off was deliberately not made here, since Section 4.3's verification is only meaningful under an exactness constraint.

The float64/Pascal-class limitation of Section 4.4 is intrinsic to shared-memory-per-block ceilings on pre-Volta architectures, not a software gap: it resolves automatically, with no code change, on compute capability 7.0 and newer. Testing on Volta, Ampere, Ada, or Hopper-generation hardware has not been performed. Larger volumes than the 100^3 case of Table 1, and volumes with different noise or structure characteristics, have not been separately benchmarked, though Section 4.3's multi-trial validation across varying noise levels and signal structures provides some evidence beyond the single configuration reported.

7. CONCLUSION

This work closes a specific, verifiable gap along two axes: a toolchain-free GPU implementation of BM4D was not previously available to the Python ecosystem as pip-installable software, and the one public GPU implementation of BM4D identified here trades algorithmic fidelity for speed in ways that prevent direct comparison to the reference package it shares a name with. This implementation is a single .py file that installs with pip, runs natively on Windows and Linux, and, as a drop-in accelerator sharing the reference package's call signature, matches its output to 0.9997 correlation while reducing wall-clock time on a 100^3 volume from 36.7-41.1 s to as little as 1.00 s. The software is released as open source and is available at the repository below.

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: *bm4d_cuda* — GPU-Accelerated BM4D Volumetric Denoising

Contact: matinmahzad@yahoo.com

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

Programming language: Python

Software required: CuPy, NumPy

Program size: ~80 KB

The source code and data are available at: <https://github.com/SeismicFlow/bm4d-cuda>

REFERENCES

- [1] K. Dabov, A. Foi, V. Katkovnik, K. Egiazarian. Image Denoising by Sparse 3-D Transform-Domain Collaborative Filtering. *IEEE Transactions on Image Processing* 16(8), 2080-2095 (2007). doi:10.1109/TIP.2007.901238
- [2] M. Maggioni, V. Katkovnik, K. Egiazarian, A. Foi. Nonlocal Transform-Domain Filter for Volumetric Data Denoising and Reconstruction. *IEEE Transactions on Image Processing* 22(1), 119-133 (2013). doi:10.1109/TIP.2012.2210725
- [3] M. Lebrun. An Analysis and Implementation of the BM3D Image Denoising Method. *Image Processing On Line* 2, 175-213 (2012). doi:10.5201/ipol.2012.l-bm3d
- [4] Y. Makinen, L. Azzari, A. Foi. Collaborative Filtering of Correlated Noise: Exact Transform-Domain Variance for Improved Shrinkage and Patch Matching. *IEEE Transactions on Image Processing* 29, 8339-8354 (2020). doi:10.1109/TIP.2020.3014721
- [5] Y. Makinen, L. Azzari, A. Foi. *bm4d* (PyPI package). <https://pypi.org/project/bm4d/>
- [6] M. Mahzad. GPU-Accelerated 3D Non-Local Means for Python: A Toolchain-Free, Pip-Installable CuPy Implementation. Manuscript submitted for publication (2026).
- [7] R. Okuta, Y. Unno, D. Nishino, S. Hido, C. Loomis. CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations. *Proceedings of the Workshop on Machine Learning Systems (LearningSys) at NeurIPS 2017* http://learningsys.org/nips17/assets/papers/paper_16.pdf (2017).
- [8] S. Sarjanoja, J. Boutellier, J. Hannuksela. BM3D Image Denoising Using Heterogeneous Computing Platforms. In: *2015 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pp. 1-8 (2015). doi:10.1109/DASIP.2015.7367257
- [9] D. Honzátko, M. Kruliš. Accelerating Block-Matching and 3D Filtering Method for Image Denoising on GPUs. *Journal of Real-Time Image Processing* 16(6), 2273-2287 (2019). doi:10.1007/s11554-017-0737-9
- [10] M. Mahmoud, B. Zheng, A. Delmás Lascorz, F. Heide, J. Assouline, P. Boucher, E. Onzon, A. Moshovos. IDEAL: Image DEnoising AcceLerator. In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50)*, pp. 82-95 (2017). doi:10.1145/3123939.3123941
- [11] V. Tananaev. *bm4d-gpu* (MIT license, software). <https://github.com/Logrus/BM4D-GPU> (2024).
- [12] R.B. Miller. Response Time in Man-Computer Conversational Transactions. *Proceedings of the AFIPS Fall Joint Computer Conference* 33, 267-277 (1968). doi:10.1145/1476589.1476628
- [13] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimeshein, L. Antiga, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems (NeurIPS)* 32, 8024-8035 (2019).