Blendtopo: A Self-Contained, Multigrid-Accelerated Topology Optimization Add-on for Blender

Philip Grünenfelder*

2026.08.22

Abstract

We describe Blendtopo, an open-source add-on that performs density-based (SIMP) structural topology optimization directly inside the 3D content-creation tool Blender. The pipeline voxelizes a user-defined build space, solves the linear-elastic state equation with a matrix-free finite-element method, and updates an element density field by the optimality-criteria method, rebuilding a closed surface after every iteration. The add-on runs entirely inside Blender on open-source code: there is no separately installed finite-element solver and no scientific-Python stack to configure. The solver uses a geometric-multigrid preconditioner whose conjugate-gradient iteration count is grid-independent over the range measured, a coarse-to-fine continuation with displacement and density warm-starts, and an optional GPU backend. We report measured crossover points for every backend choice the add-on makes automatically.

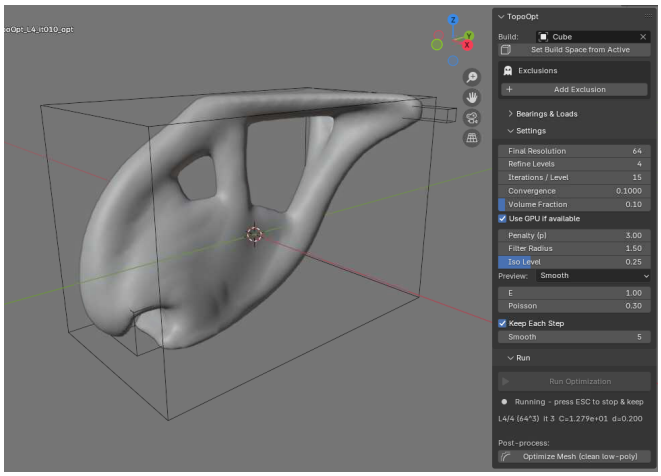

Figure 1: The Blendtopo panel inside Blender, with an optimized structure forming inside the wireframe build space. Bearings, loads and exclusions are ordinary mesh objects; the result is rebuilt after every iteration.

1 Introduction

Topology optimization (TO) distributes a limited amount of material within a design domain so as to maximize structural performance, typically minimizing compliance (maximizing stiffness) under a volume constraint [2]. The idea that an optimal load path is a well-posed mathematical object predates numerical methods by a century: Michell [1] derived the least-material frame layouts that still serve as analytical benchmarks for modern solvers. The Solid Isotropic Material with Penalization (SIMP) method, combined with sensitivity filtering and the optimality-criteria (OC) update, is the standard educational and practical realization of that idea [3, 4, 5]. We treat the structural, single-physics case throughout; TO is applied equally to thermal, fluid and multiphysics objectives, and to formulations other than density-based SIMP, for which we refer to the comparative review of Sigmund and Maute [6].

We embed a complete 3D SIMP pipeline inside Blender, a widely used free modelling tool. The motivation is accessibility: a user who already models a part in Blender can specify a build space, supports and loads with familiar mesh objects and obtain an optimized structure without leaving the application, installing a separate FE solver, or configuring a scientific-Python stack. The add-on is implemented in pure Python over Blender's bundled NumPy and built-in mesh modifiers.

This note documents the pipeline and the engineering choices that make it practical inside an interactive modelling tool. We make no claim of algorithmic novelty in the optimization formulation itself; the contribution is a careful, self-contained integration, and a measured account of where each backend the add-on can choose actually pays off — including the two cases where the apparently faster hardware path is the slower one.

Figure 1 shows the workflow. The build space, bearings, loads and exclusions are ordinary Blender objects, selected in the panel alongside the SIMP parameters — volume fraction, penalization, filter radius and resolution — so the entire setup is done with modelling tools the user already knows.

2 Related work

The SIMP/OC formulation we implement is the one established by Bendsøe and Sigmund [2] and made widely reproducible by the compact MATLAB codes of Sigmund [3] and Andreassen et al. [4]. Of these, the 3D code of Liu and Tovar [5] is the closest published antecedent of our implementation: our element loop follows the same structure, with the assembled sparse system replaced by a matrix-free operator (Section 3.3). Filtering the sensitivities to remove checkerboarding and impose a length

*B.Sc. Mechanical Engineering, M.Sc. Business Engineering.

scale likewise follows standard practice; Bourdin [7] gives the analysis of these filters.

Using multigrid as a preconditioner for the CG solve inside SIMP is likewise established rather than new. Amir, Aage and Lazarov [8] analyze multigrid-CG for TO and show the grid-independent iteration behaviour we reproduce in Section 4.1; Wu, Dick and Westermann [9] pair a matrix-free multigrid solver with a GPU to push high-resolution TO onto a single desktop, which is the regime our GPU backend targets; and the PETSc framework of Aage et al. [10] applies the same ideas at cluster scale. Our solver is a deliberately modest instance of that family, sized for one workstation and implemented over Blender's bundled NumPy.

What distinguishes Blendtopo is therefore not the algorithm but the packaging. Most TO capability reaches users either as standalone engineering code [10] or as a feature of a commercial CAD/CAE suite. The closest prior work in the free-tool space is ToOptix [11], which provides topology optimization add-ons for both Blender and FreeCAD over a shared core; there the finite element solution is performed by an external CalculiX installation driven through .inp files, so a working setup spans the modelling application and a separately installed solver. Blendtopo instead carries its own matrix-free FE solver, multigrid preconditioner and surface extraction inside the extension package, so the entire loop — voxelize, solve, update, remesh — runs in-process inside Blender. Self-containment is also a platform requirement: a Blender extension may not depend on a package the user is expected to install separately.

3 Method

3.1 Problem setup and voxelization

The user supplies four kinds of mesh objects: a build space (the domain to be filled), optional exclusions (keep-out regions), bearings (supports) and loads (applied forces). The build space's world-space axis-aligned bounding box is discretized into a regular grid of trilinear hexahedral (Hex8) elements; the grid is centred on the box. Membership of each voxel centre and grid node is determined by a ray-parity point-in-mesh test against a BVH of each input mesh. To avoid the well-known degeneracy of axis-aligned rays grazing coplanar faces, the test casts a single slightly tilted, non-axis-aligned ray, which removes the scattered misclassifications that otherwise puncture the mask. Nodes inside bearing meshes contribute fixed degrees of freedom (selectable per axis); voxels inside exclusions are removed. Each load object carries one force vector, which is divided equally among the nodes falling inside its mesh, so the applied resultant is independent of grid resolution.

3.2 SIMP optimization

With element densities $\rho_e \in [0,1]$ the Young's modulus is interpolated as $E_e = E_{\min} + \rho_e^p\,(E_0 - E_{\min})$ with penalization p. We minimize compliance $c(\rho) = \mathbf{u}^\top \mathbf{K}(\rho)\mathbf{u}$ subject to a volume fraction constraint, where $\mathbf{u}$ solves $\mathbf{K}(\rho)\mathbf{u} = \mathbf{f}$. Sensitivities $\partial c/\partial \rho_e = -p\,\rho_e^{p-1}(E_0 - E_{\min})\,\mathbf{u}_e^\top \mathbf{k}_0 \mathbf{u}_e$ are smoothed with the classical sensitivity filter of radius $r_{\min}$ (measured in elements, and setting the minimum feature size), and densities are updated by the optimality-criteria scheme with a bisection on the Lagrange multiplier [3, 4]. Default values for p, $r_{\min}$, E_0, $E_{\min}$, Poisson's ratio and the iteration budget are listed in Appendix A.

3.3 Matrix-free finite elements

All elements share a single Hex8 stiffness matrix $\mathbf{k}_0$, formed via 2-point Gauss quadrature—the minimum order to integrate trilinear terms exactly and prevent spurious hourglass modes. Because $\mathbf{k}_0$ is shared, the operator $\mathbf{Ku}$ is applied element-by-element without assembling a global sparse matrix: gather the 24 element displacements, multiply by $\mathbf{k}_0$, scale by E_e, and scatter-add to the global residual. In both solvers described below the linear system is solved by preconditioned conjugate gradients (PCG).

The two solvers treat the empty part of the bounding box differently. The single-grid solver assembles its system only over the degrees of freedom touched by build-space elements, so a part that occupies a fraction of its bounding box yields a correspondingly smaller system. The multigrid solver cannot do this — a coarsening hierarchy needs the full regular grid — so it carries the void as elements of modulus $E_{\min}$ and solves the larger system. It wins anyway: the DOF saving is bounded by how much margin the bounding box contains, whereas the grid-independent iteration count is worth an order of magnitude at the resolutions of Section 4.1. The single-grid path therefore remains in use only where a hierarchy cannot be built at all — grids that are too small or have odd dimensions — in which case the solver falls back to single-level Jacobi-PCG.

3.4 Geometric multigrid preconditioner

We use a matrix-free geometric-multigrid V-cycle as the preconditioner [8]: a hierarchy of 2×-coarsened grids, weighted-Jacobi smoothing, trilinear prolongation with its transpose as restriction, and rediscretized coarse operators (densities averaged over child elements, element stiffness scaled by the coarse element size). Crucially, the outer PCG always applies the true fine-grid operator for its matrix–vector product and residual, so the computed displacement is correct regardless of preconditioner quality; the V-cycle only affects the iteration count.

3.5 Continuation and warm starts

Optimization proceeds coarse-to-fine: the density field is solved on a coarse grid and trilinearly prolongated to seed the next finer grid. Within a grid, each PCG solve is warm-started from the previous iteration's displacement, and the first solve of each finer grid is warm-started from the prolongated coarse displacement. The CG tolerance is loosened while the design is changing quickly and tightened as it settles. Coarse levels stop early once the maximum per-iteration density change falls below a threshold; the finest level runs to its full iteration budget.

3.6 Output and integration

After every iteration the density field is converted to a triangle/quad surface by the naive variant of Surface Nets [12, 13]: each straddling cell contributes one vertex placed at the mean of that cell's edge crossings. Gibson's original algorithm then relaxes those vertex positions iteratively, subject to a constraint keeping each vertex inside its own cell, which yields a smoother surface; we omit that step, trading a little smoothness for a single fully vectorized pass with no iterative solve. Padding the field with an empty border closes the shell where the structure meets the domain boundary.[1] The optimization runs on a background thread so the viewport stays interactive, with results written to a dedicated collection and the previous step hidden so the shape appears to morph in place. A GPU backend (via CuPy) is available optionally.[2]

4 Results

4.1 Solver scaling

Table 1 reports PCG iterations to a relative residual of 10^{-6} for a solid cantilever ($\rho = 1$ everywhere) at four resolutions, comparing the diagonal preconditioner with the multigrid V-cycle (CPU, NumPy). The multigrid iteration count is effectively constant (≈ 10) as the problem grows by an order of magnitude in degrees of freedom, while the diagonal count grows roughly linearly with the number of elements per edge. Both solvers converge to the same displacement (relative difference $< 10^{-8}$); the multigrid V-cycle is a preconditioner only and does not change the solution.[3]

[1] Our regression tests assert that the extracted surface has no boundary edges (no edge used by exactly one face), including for structures touching the domain boundary. Note that one vertex per cell does not by itself guarantee a manifold surface: a cell straddled by two disjoint sheets yields a single shared vertex.

[2] It is enabled only after a startup self-test confirms it reproduces the CPU solution; otherwise the CPU path is used transparently.

[3] Two scope notes. This is the uniform-density case, which is the easiest for the V-cycle; iteration counts on the high-contrast fields arising mid-optimization are not characterized here. And this count is for the inner linear solve at a fixed density field, not the number of outer SIMP design iterations, which is much larger (Section 4.8).

Grid	DOF	Jacobi-PCG	MG-PCG
$12 \times 6 \times 6$	1 911	85	11
$16 \times 8 \times 8$	4 131	114	10
$24 \times 12 \times 12$	12 675	172	11
$32 \times 16 \times 16$	28 611	228	10

Table 1: PCG iterations to relative residual 10^{-6} for a solid cantilever. Diagonal (Jacobi) preconditioning versus the geometric-multigrid V-cycle.

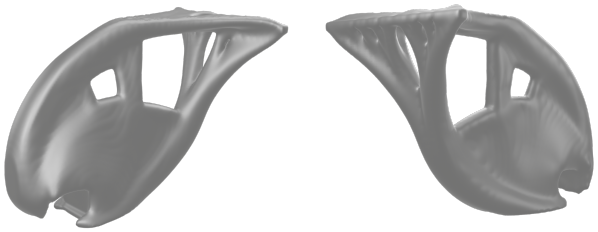

Figure 2: An optimized part produced by Blendtopo (two views), fully created and rendered in Blender.

Figure 2 shows a representative output: a branching, organic load path with no residual staircasing from the underlying voxel grid, produced by the naive Surface Nets reconstruction described in Section 3.6 and rendered without post-processing outside Blender.

4.2 Validation against beam theory

The FEM solver alone, with $\rho = 1$ everywhere and no SIMP involved, is validated against an independent analytical result. A $16 \times 6 \times 6$ solid cantilever of unit voxels ($E_0 = 1$, $\nu = 0.3$, $L/h \approx 2.7$) is fully fixed at its root face and loaded by a unit downward force at each of the 7 nodes along the tip edge (resultant $P = 7$). The computed tip deflection is 97.18, against 98.20 from Timoshenko beam theory (Euler–Bernoulli bending plus a first-order shear correction, $k = 5/6$) — an agreement of 1.0%.[4] There is no closed-form solution for an optimized topology, so this check is deliberately performed on the solid, pre-SIMP case only.

4.3 Warm starts

Seeding a fine level from a prolongated coarse result measurably outperforms a cold start for the same small number of fine iterations. In our regression test, an $8 \times 4 \times 4$ cantilever optimized for 15 iterations and prolongated onto a $16 \times 8 \times 8$ grid reaches compliance 2.04×10^3 after

[4] The shear term is what matters at this aspect ratio: plain Euler–Bernoulli theory predicts 88.49 and so underpredicts the deflection by 9.9%, which the solver correctly does not reproduce. The numerical core is additionally covered by headless regression tests: monotone compliance reduction under a held volume fraction, density and displacement resampling, and the no-boundary-edge check of Section 3.6.

Grid	DOF	CPU [s]	GPU [s]	Speedup
$12 \times 6 \times 6$	1 911	0.026	0.471	0.05
$16 \times 8 \times 8$	4 131	0.028	0.342	0.08
$24 \times 12 \times 12$	12 675	0.103	0.452	0.23
$32 \times 16 \times 16$	28 611	0.213	0.398	0.54
$48 \times 24 \times 24$	91 875	0.846	0.492	1.72
$64 \times 32 \times 32$	212 355	1.812	0.582	3.11
$80 \times 40 \times 40$	408 483	4.782	1.076	4.45

Table 2: Mean wall-clock time per SIMP outer iteration (CPU/NumPy vs. single GPU/CuPy, 36 repeats), and the CPU/GPU speedup ratio (>1 means GPU faster), on an Intel Haswell-class CPU and one NVIDIA GTX 970.

5 fine iterations, against 5.17×10^3 for a cold start given the same 5 iterations — a factor of 2.5. The build-space DOF restriction (Section 3.3) separately shrinks the linear system whenever the part occupies only part of its bounding box.

4.4 CPU vs. GPU pipeline timing

We measured the four pipeline stages (voxelization, matrix/FEA setup, multigrid-hierarchy setup, and the per-iteration solve) separately at seven grid resolutions, on both the CPU (NumPy) and GPU (CuPy) backends, on a machine with an Intel Haswell-class CPU (4 logical cores) and an NVIDIA GeForce GTX 970 (CuPy 14.0.1, NumPy 2.1.3, Windows 11); 36 repeats per stage per resolution. Of the four stages, only solve_iter (one PCG solve plus strain-energy, filter and OC update) recurs every SIMP iteration and dominates total run time at every resolution tested; the three setup stages are paid once per optimization run and were, on this hardware, essentially identical between backends (e.g. matrix setup at 91 875 DOF: 30.2 ms CPU vs. 28.8 ms GPU), since none of that one-time staging work is accelerated differently by either array library. Table 2 and Figure 3 report solve_iter alone.[5]

The GPU is markedly slower than the CPU below ∼50 000 DOF — kernel-launch and host/device transfer overhead per PCG iteration dominates at these problem sizes — and overtakes the CPU beyond that, with the advantage widening steadily up to the largest grid tested (408 483 DOF, 4.45×); log–log interpolation between the two bracketing resolutions places the crossover at ≈53 400 DOF. The GPU backend is therefore a net loss at every resolution in Table 1 and only pays off on substantially larger build spaces. Both the crossover point and the absolute times are likely specific to this GPU.

Figure 3 isolates solve_iter because it is the stage that actually differs by backend, but that leaves open how much of a whole optimization run it accounts for. Figure 4 reconstructs a full-run time budget per grid size and backend as voxelize + matrix_setup + multigrid_setup + 40×solve_iter, using 40 (the default max_iter) as a fixed, backend-independent iteration count so the comparison stays fair (read ratios from it, not absolute totals). The one-time voxelization and FEA/MG setup stages (grey) are visually reduced to a thin sliver at every grid size, on both backends, contributing well under 1% of total run time even at the largest grid tested. The optimization loop (colored) is what balloons, and it does so very differently per backend: on the CPU the reconstructed total grows 187× across this range (1.0 s to 192 s, slightly sub-cubic in elements-per-edge), whereas on the GPU the same range costs only 2.3× more, because the fixed per-iteration kernel-launch and transfer overhead that makes small GPU runs so wasteful (top two bars, GPU 12–18× slower overall than CPU) is amortized away at the largest sizes, where the reconstructed total is 4.4× faster on the GPU. In conclusion, the setup stages never justify picking a backend either way.

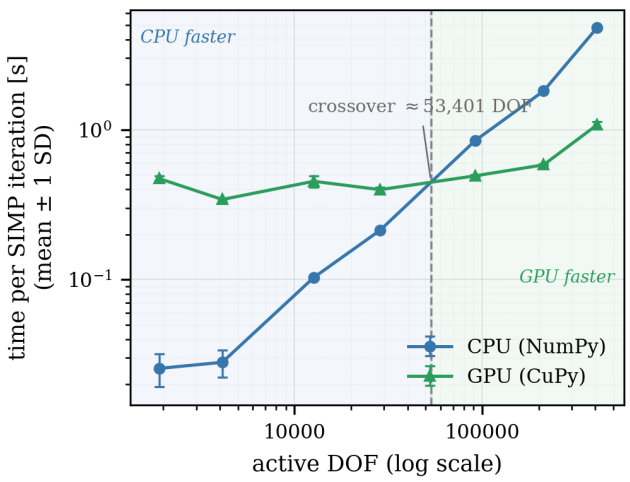

Figure 3: Mean time per SIMP iteration vs. active DOF, CPU (NumPy) vs. single GPU (CuPy) (data of Table 2). Error bars are ±1 SD over 36 repeats; shading marks which backend is faster, and the dashed line marks the interpolated crossover (≈53 400 DOF).

4.5 Two GPUs, two problems: concurrent throughput

Two distinct questions arise once a second card is present: whether two independent optimizations can be run concurrently, one per card, and whether a single optimization can be split across both. This section answers the first; Section 4.6 answers the second, and the two results point in opposite directions.

Alongside the solo measurements of Table 2, we ran the same resolution ladder as two independent problems executing concurrently, one pinned to each GTX 970, at 36

[5] All benchmark cantilevers here are solid, so the active DOF plotted in the figures equals the total grid DOF listed in the tables; the two differ only when a part occupies part of its bounding box.

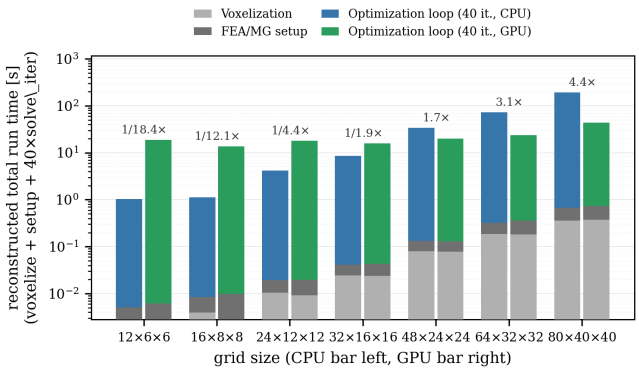

Figure 4: Reconstructed total run time per grid size, CPU (left bar) vs. single GPU (right bar), stacked by pipeline stage. Log-scale y-axis; annotations give the CPU/GPU ratio of the reconstructed total per size.

Grid	DOF	Solo [s]	Per-job	Thrpt.
$12 \times 6 \times 6$	1 911	0.471	1.23×	1.63×
$16 \times 8 \times 8$	4 131	0.342	1.28×	1.56×
$24 \times 12 \times 12$	12 675	0.452	1.34×	1.50×
$32 \times 16 \times 16$	28 611	0.398	1.40×	1.43×
$48 \times 24 \times 24$	91 875	0.492	1.16×	1.72×
$64 \times 32 \times 32$	212 355	0.582	1.09×	1.84×
$80 \times 40 \times 40$	408 483	1.076	1.05×	1.91×

Table 3: Two independent optimizations run concurrently, one per GTX 970, vs. one running alone. "Per-job" is the slower device's mean solve_iter relative to solo; "Thrpt." is the resulting aggregate throughput, where 2× would be ideal.

repeats per stage per device. Table 3 reports the per-job cost and the resulting aggregate throughput. Contention is worst at small grids, where each job is latency-bound and the two processes compete for host-side scheduling rather than for device compute: per-job cost rises up to 1.40× at 28 611 DOF, so two cards deliver only 1.43× the work of one. As the problem grows the picture improves monotonically — at 408 483 DOF each job pays just 5% over its solo time and aggregate throughput reaches 1.91×, close to the ideal 2×. Running independent jobs concurrently is therefore a sound use of a second card, and becomes more efficient exactly where run times are long enough for it to matter.

4.6 Splitting one solve across two GPUs

The opposite strategy — partitioning a single problem's element list across every visible CUDA device and summing the partial **Kp** products — behaves the other way round. We repeated a ten-resolution ladder (5 repeats each, single session) forcing compute_mode="GPU" against "MULTI_GPU" on the same two GTX 970s. Table 4 and Figure 5 show that two GPUs are faster only below ≈45 000 DOF (up to 2.3× at 4 131 DOF), and increasingly slower above it, reaching 0.28× (roughly 3.6× slower) at 1.10M DOF.

Grid	DOF	1 GPU [s]	2 GPUs [s]	Ratio
$12 \times 6 \times 6$	1 911	0.403 ± 0.034	0.188 ± 0.019	2.14
$16 \times 8 \times 8$	4 131	0.314 ± 0.011	0.137 ± 0.001	2.29
$24 \times 12 \times 12$	12 675	0.448 ± 0.023	0.242 ± 0.015	1.85
$32 \times 16 \times 16$	28 611	0.388 ± 0.002	0.292 ± 0.007	1.33
$48 \times 24 \times 24$	91 875	0.592 ± 0.013	0.922 ± 0.019	0.64
$64 \times 32 \times 32$	212 355	0.674 ± 0.002	1.538 ± 0.084	0.44
$80 \times 40 \times 40$	408 483	1.253 ± 0.018	3.892 ± 0.148	0.32
$96 \times 48 \times 48$	698 691	1.819 ± 0.015	5.927 ± 0.195	0.31
$112 \times 56 \times 56$	1.10M	2.989 ± 0.010	10.506 ± 0.160	0.28
$128 \times 64 \times 64$	1.64M	3.123 ± 0.033	9.567 ± 0.032	0.33

Table 4: Mean wall-clock time per SIMP outer iteration ± 1 SD, single GPU vs. both GPUs splitting the same problem (5 repeats, one session), on two NVIDIA GTX 970s. Ratio is 1-GPU/2-GPU time (>1 means 2 GPUs faster). See Appendix B on reproducibility.

The suspected culprit is data movement, not compute. The pool broadcasts the entire solution vector to every device on every call (host-staged, no peer-to-peer) and each device returns a full ndof-length partial result summed on the host, so splitting the element list halves each card's compute without reducing what it receives or returns. Compounding this, the parallel plan keeps CG and V-cycle vector state on the host rather than resident on a device, so every matvec per V-cycle pays a fresh round trip to both cards. Below the crossover this cost is dominated by fixed per-call latency and two latency-bound kernels genuinely overlap; above it, the doubled data movement grows with problem size while the compute saved per card stays a constant fraction.

This might suggest selecting the multi-GPU path adaptively, below the crossover only. It does not, because that region is already owned by a third backend: wherever two GPUs beat one, the CPU beats both. At 28 611 DOF the CPU needs 0.213 s against 0.292 s for two GPUs and 0.388 s for one; at 12 675 DOF, 0.103 s against 0.242 s. There is thus no problem size at which splitting a solve across two cards is the best available choice, and Blendtopo's automatic backend selection never chooses it.

4.7 Voxelization backend: vectorized ray casting vs. a bundled spatial index

The point-in-mesh test used during voxelization (Section 3.1) has two implementations of the same ray-parity (crossing-number) algorithm: a dependency-free variant that vectorizes the ray–triangle intersection over all query points and all mesh triangles at once as NumPy array operations, and an accelerated variant that first narrows the candidate triangles per ray with a bounding-volume hierarchy from trimesh [14] built on an rtree [15] spatial index before doing the same intersection test.

We benchmarked the two backends in a controlled, paired, repeated-measures experiment: closed icosphere

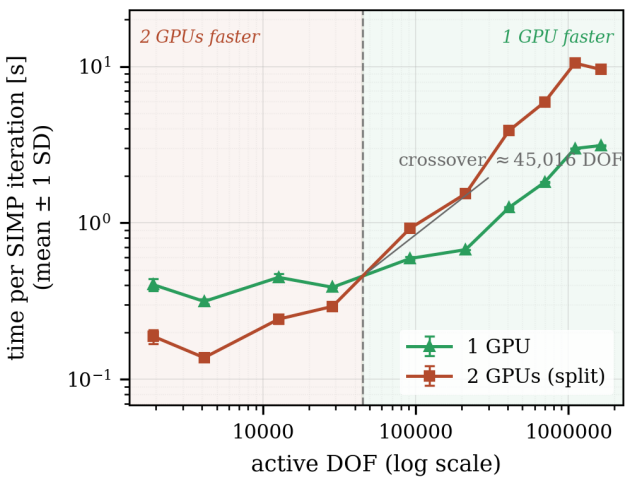

Figure 5: Mean time per SIMP iteration vs. active DOF, single GPU vs. both GPUs splitting one problem (data of Table 4). Note the direction: two GPUs win only below the crossover (≈45 000 DOF) and lose by a widening margin above it — the opposite pattern from Figure 3.

test meshes at seven triangle counts (a geometric ladder from 20 to 81 920 triangles), 8–20 independent repeats per level with freshly sampled query points each time, and a paired t-test between the two backends' wall-clock time at every level. Table 5 and Figure 6 summarize the result: the spatial-index backend carries a fixed overhead (constructing the mesh object and the index) that makes it clearly slower than the direct NumPy test below a few hundred triangles. At 320 triangles the two backends are statistically indistinguishable (mean speedup 0.81 ± 0.27, straddling parity); log–log interpolation between that level and the next (1 280 triangles) places the break-even point at ≈551 triangles. From 1 280 triangles up the spatial-index backend is faster with high statistical significance ($p < 0.02$, and $p < 10^{-4}$ above ∼5 000 triangles). The two backends agreed on all 105 trials' inside/outside classification (bitwise-identical masks in every case), confirming the accelerated path is a performance optimization only. Based on this result the add-on routes to the spatial-index backend once an input mesh exceeds 600 triangles and uses the NumPy path below that.

4.8 Cost of the coarse-to-fine stage ladder

Table 6 isolates the cost of continuation directly, timing the SIMP+FEA solver alone (no voxelization or meshing) for a synthetic cantilever at a fixed total iteration budget split evenly across 1–4 coarse-to-fine stages, built the way Blendtopo's own operator builds them (geometric halving down to a floor of 4 elements). Every configuration runs the exact same total iteration count for a given budget — 24 and 36 were chosen because they are divisible by 1, 2, 3 and 4, so integer division never

Tris	n	NumPy [s]	Trimesh+rtree [s]	Speedup
20	20	0.041 ± 0.014	0.161 ± 0.068	0.27 ± 0.09
80	20	0.089 ± 0.039	0.238 ± 0.050	0.39 ± 0.24
320	20	0.277 ± 0.105	0.340 ± 0.036	0.81 ± 0.27
1 280	15	0.870 ± 0.279	0.663 ± 0.228	1.38 ± 0.51
5 120	12	1.999 ± 0.318	1.199 ± 0.226	1.69 ± 0.28
20 480	10	3.708 ± 0.317	2.135 ± 0.093	1.74 ± 0.17
81 920	8	3.794 ± 0.382	2.493 ± 0.193	1.53 ± 0.20

Table 5: Inside-test wall-clock time (mean ± 1 SD over n paired trials) for the NumPy and bundled trimesh+rtree backends, and their paired speedup ratio, across a geometric ladder of icosphere triangle counts. Differences from 1 280 triangles up are statistically significant (paired t-test, $p < 0.02$); at 320 triangles the two backends are indistinguishable.

Budget	Stages (el./edge)	Wall (s)	Speedup
24	48	35.06	1.00×
24	24/48	14.58	2.40×
24	12/24/48	11.08	3.16×
24	6/12/24/48	12.85	2.73×
36	48	39.07	1.00×
36	24/48	15.48	2.52×
36	12/24/48	13.97	2.80×
36	6/12/24/48	15.22	2.57×

Table 6: Wall-clock cost of a fixed total SIMP-iteration budget, split evenly across 1–4 coarse-to-fine stages, isolating the SIMP+FEA solver (synthetic cantilever, 48 × 24 × 24 finest grid, 91 875 DOF). Every row runs its full budget; speedup is relative to the 1-stage (finest-grid-only) run at the same budget.

rounds a stage count down — and the speedup column is therefore purely the cost of splitting a fixed amount of optimization work into cheaper stages.

At 24 total iterations, spreading them over 3 stages (12/24/48) instead of running all 24 at the finest grid is 3.2× faster in wall-clock terms; at 36 iterations the best case (again 3 stages) is 2.8× faster. The mechanism is geometric: each 2× coarsening is an 8× reduction in DOF, so most of a coarse-to-fine run's iterations are spent on grids that are cheap by construction, independent of the target part's shape. In both budgets the 4-stage ladder is slightly slower than the 3-stage one — once a coarse stage's own cost is negligible, adding another is pure overhead without shrinking the dominant finest-grid stage, so the DOF argument does not imply that more stages are always better.

5 Limitations and future work

5.1 Continuation and solution-space locality

Because each finer level is seeded from the prolongated result of the previous, coarser level (Section 3.5), the density field is not found by an unconstrained search

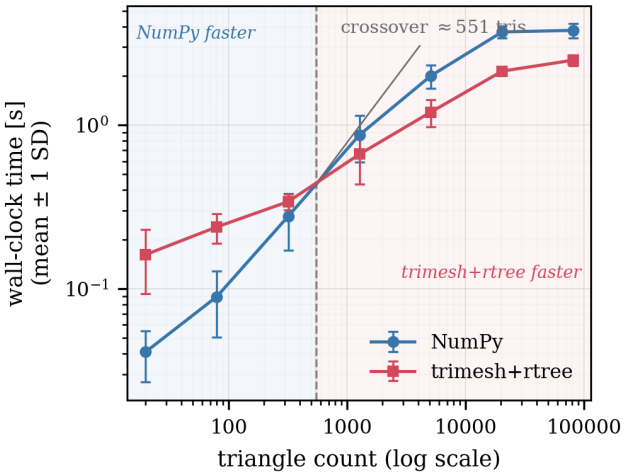

Figure 6: Mean inside-test wall-clock time vs. triangle count for the NumPy and bundled trimesh+rtree backends (data of Table 5). Error bars are ±1 SD across repeated, independently sampled trials; shading marks the triangle-count range in which each backend is faster, and the dashed line marks the interpolated crossover (≈551 triangles).

of the fine-grid solution space; it is found by walking a path that starts inside the much smaller solution space of a coarse grid and is refined step by step. Every intermediate design therefore lies in the neighborhood of a structure that already worked at some coarser resolution. That is exactly why the scheme is fast — and it also means the continuation implicitly favors final topologies that remain plausible at coarser resolutions, rather than exploring the fine-grid design space on equal footing. A single-resolution run given the same total budget is not constrained by any coarser stage and can settle into a different, more locally intricate, local optimum.

We observed this qualitatively on a test part (Fig. 7, Fig. 8), comparing runs that spend their iterations either entirely at the finest resolution or split evenly across 2, 3 or 4 coarse-to-fine stages. Run only at the finest resolution, the structure needs roughly 40 iterations to settle into a filigree topology; a 24/48, 12/24/48 or 6/12/24/48 continuation reaches a visually converged, comparably stiff structure in as few as 30 total iterations, at a small but systematic reduction in maximum surface detail. We have no formal argument that continuation preserves global optimality — compliance TO is nonconvex regardless of solver — so this should be read as an empirically effective heuristic rather than a guarantee.

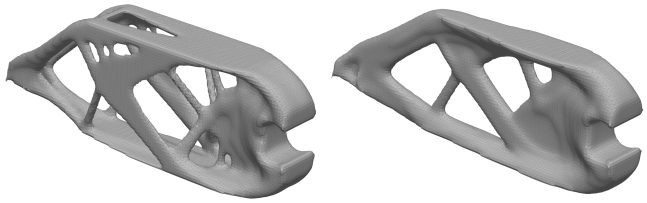

Figure 7: Same part and total iteration budget (40 iterations), one coarse-to-fine stage vs. four. Left: single stage at 48 el./edge. Right: stages 6/12/24/48 el./edge, 40 iterations total. The single-stage run resolves finer struts; the 4-stage run reaches an optimized topology faster, but a less filigree one.

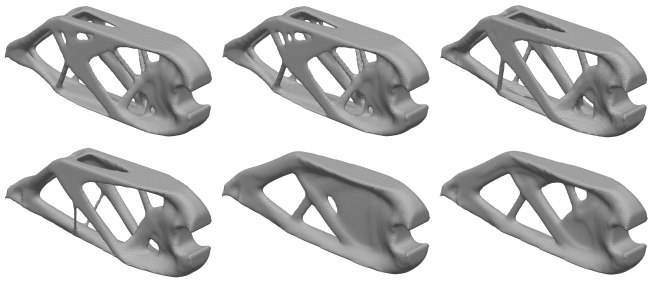

Figure 8: Six runs of the same test part, in el./edge. Left to right, top to bottom: 48 (30 it); 48 (40 it); 24/48 (30 it); 12/24/48 (30 it); 6/12/24/48 (28 it); 6/12/24/48 (40 it). The iteration-matched comparisons are the four 30-iteration runs and the two 40-iteration runs. Fewer stages reach finer surface detail; more stages converge to a stable topology in fewer total iterations.

5.2 Scope

The current formulation minimizes compliance for static linear elasticity with a single load case (multiple applied loads are combined into one case) and an isotropic material. The voxelization benchmark of Section 4.7 used synthetic icosphere meshes of controlled triangle count; real build-space, exclusion and bearing meshes may differ in triangle-to-volume ratio and concavity, which could shift the measured crossover somewhat, though we expect triangle count to remain the dominant factor for both backends' cost. All timing results come from a single machine, so the reported crossovers are properties of that hardware rather than universal constants.

6 Availability

Blendtopo is released under the GPL-3.0 license and installs as a native Blender (4.2+) extension. Source, releases and documentation are available in the project repository.

A Default parameters

Table 7 lists the defaults used throughout this note. All quantities are nondimensional; the voxel edge is the unit of length, so $r_{\min}$ is measured in elements. Because $E_0 = 1$, the reported compliances are a stiffness ranking in nondimensional units rather than a stress-based verification.

Symbol	Quantity	Value
E_0	Young's modulus, solid	1.0
$E_{\min}$	Young's modulus, void	10^{-9}
ν	Poisson's ratio	0.3
p	SIMP penalization	3.0
$r_{\min}$	Sensitivity filter radius	1.5 el.
—	Volume fraction	0.3
—	Max. SIMP iterations	40
—	Outer convergence tol.	0.01
—	PCG relative residual (Table 1)	10^{-6}

Table 7: Default SIMP and solver parameters.

B Measurement notes

Two notes about Table 4.

Is the multi-GPU degradation real, or noise? The standard deviations within that session are mostly under 3% of the mean, and the effect spans more than an order of magnitude, so the trend is robust within the session. Two irregularities are visible: the ratio at 1.64M DOF (0.33×) is slightly better than at 1.10M (0.28×), so the degradation is not perfectly monotone at the top end.

Why do the single-GPU times differ between Table 2 and Table 4? The two tables come from separate benchmarking sessions with different repeat counts, and disagree by up to 16% (e.g. 1.076 s vs. 1.253 s at 408 483 DOF). We report both as measured rather than reconciling them; that spread is a fair empirical bound on run-to-run reproducibility on this hardware, and it is smaller than every effect we claim.

Acknowledgments

The design, implementation, benchmarking and drafting of this work were carried out with the assistance of Anthropic's Claude.[6] All results reported here were produced and verified by the author.

[6] https://www.anthropic.com/claude

References

[1] A.G.M. Michell. The limits of economy of material in frame-structures. Phil. Mag., Series 6, 8(47):589–597, 1904.

[2] M.P. Bendsøe and O. Sigmund. Topology Optimization: Theory, Methods and Applications. Springer, 2003.

[3] O. Sigmund. A 99 line topology optimization code written in Matlab. Struct. Multidisc. Optim., 21:120–127, 2001.

[4] E. Andreassen et al. Efficient topology optimization in MATLAB using 88 lines of code. Struct. Multidisc. Optim., 43:1–16, 2011.

[5] K. Liu and A. Tovar. An efficient 3D topology optimization code written in Matlab. Struct. Multidisc. Optim., 50:1175–1196, 2014.

[6] O. Sigmund and K. Maute. Topology optimization approaches: A comparative review. Struct. Multidisc. Optim., 48:1031–1055, 2013.

[7] B. Bourdin. Filters in topology optimization. Int. J. Numer. Methods Eng., 50(9):2143–2158, 2001.

[8] O. Amir, N. Aage and B.S. Lazarov. On multigrid-CG for efficient topology optimization. Struct. Multidisc. Optim., 49:815–829, 2014.

[9] J. Wu, C. Dick and R. Westermann. A system for high-resolution topology optimization. IEEE Trans. Vis. Comput. Graph., 22(3):1195–1208, 2016.

[10] N. Aage, E. Andreassen and B.S. Lazarov. Topology optimization using PETSc: An easy-to-use, fully parallel, open source topology optimization framework. Struct. Multidisc. Optim., 51:565–572, 2015.

[11] ToOptix: open-source topology optimization addons for Blender and FreeCAD. https://github.com/Foxelmanian/ToOptix, accessed 2026-08-22.

[12] S.F.F. Gibson. Constrained elastic surface nets: generating smooth surfaces from binary segmented data. MICCAI, 1998.

[13] M. Lysenko. Smooth Voxel Terrain (Part 2): Naive Surface Nets, 0fps blog, July 12, 2012. http://0fps.net/2012/07/12/smooth-voxel-terrain-part-2/, accessed 2026-07-04.

[14] M. Dawson-Haggerty et al. trimesh (software), 2019. https://trimesh.org, accessed 2026-07-04.

[15] S. Gillies et al. Rtree: Spatial indexing for Python, 2024. https://rtree.readthedocs.io, accessed 2026-07-04.