ChatGPT

GPU-Parallel Patterns for a Voxel-Based Simulation Engine

Implementing a *voxel-based cellular-physics engine* in a single GPU pass requires rethinking the simulation as a **massively parallel, local update** problem. Modern approaches borrow ideas from cellular automata (CA), convolutional neural nets, and differentiable physics. Below we outline concrete architecture patterns and PyTorch coding strategies to achieve a unified update step handling gravity, lateral flow, and absorption for multiple materials – all without Python loops.

1. Multi-Channel Tensor World Representation

Use a **multi-channel tensor** to represent the world grid, where each channel corresponds to a material or property field. For example, a 2D world can be a tensor of shape `(C, H, W)` (or `(batch, C, H, W)` in PyTorch), and a 3D world `(C, D, H, W)`. Each cell's state is a vector of material quantities (e.g. amount of air, water, sand, etc.). This allows per-material parameters and interactions to be encoded as operations on channels. In such a discretized world, **space and time are uniform** and local – ideal for parallel GPU updates [1] [2]. Multi-material interactions (e.g. water soaking into soil) can occur **in-place** via channel transfers, rather than needing separate data structures. Notably, adding more material types doesn't significantly slow the simulation – handling multiple materials in one CA adds *minimal overhead* while yielding complex behavior [3].

Tips:
- If certain materials need extra state (e.g. "wetness" of soil), you can dedicate a channel for that property. Keep all state in a single tensor for co-located memory access.
- Consider normalizing or bounding the sum of materials per cell (especially if modeling incompressible vs compressible materials). For example, incompressible materials might be capped at 1.0 per cell, whereas compressible gas could exceed 1.0. This "volume capacity" concept can be encoded as part of the update rules.

2. Stencil-Based, Single-Pass Updates via Convolution

Express the physics as a stencil operation – each cell's new state is a function of its local neighborhood. This is analogous to a convolutional layer in CNNs, where the kernel covers the neighboring voxels. By designing the update as convolution(s), we leverage GPU parallelism: all cells compute their new state simultaneously from neighbors [4]. In fact, many CA and PDE updates can be formulated as convolution filters (plus nonlinearities) and executed via deep-learning libraries [5] [6]. For example, Conway's Game of Life can be implemented with a 3×3 convolution to sum neighbors [5], and custom physics can use learned or hand-crafted kernels. In our context, we use convolution *not to learn features*, but as a **GPU-optimized stencil computation** of neighbor interactions.

Approach: Define small convolution kernels (e.g. 3×3 in 2D, or 3×3×3 in 3D) that pick values from neighbor cells and compute tentative transfers of material. You may use **depthwise/grouped convolution** so that each material (channel) has its own kernel or parameters [7]. For instance, a depthwise conv with groups=C can apply distinct weights per material for how it spreads to each neighbor. Cross-channel kernel weights can implement conversions (e.g. water turning into "wet soil" in a neighbor cell). The conv output will produce the *change* to apply to each cell's channels.

- *Gravity:* can be encoded by a kernel that has a weight encouraging downward flow. E.g. a 3×3 kernel for sand might have a positive weight for the cell directly below and negative weight for the cell itself, causing a downward push [8]. This essentially computes something like Δ_sand = `k_down*sand_above + k_self*sand_here` aggregated over neighbors, mimicking a finite-difference of "sand density" in the vertical direction. A large negative weight at the cell and positive below yields movement downward if below is emptier (the nonlinearity will handle the "if empty" condition, as described next).

- *Lateral Dispersion:* use neighboring side weights to diffuse materials sideways. For fluids, a kernel can take a fraction of the cell's water and distribute it to left/right neighbors if they have less water. In practice this might be achieved by computing neighbor differences and applying a **ReLU** (rectifier) as a non-linear step (only flow out if you have more than neighbor). This can be done by *post-processing the conv results* – e.g. after a conv computes raw "flow" suggestions, clamp negative flows to zero to ensure one-directional flow.

- *Absorption:* is inherently a cross-material interaction. You can handle it by **multi-output convolution** or sequential operations. For example, to model soil absorbing water: in the conv kernel for water, include a weight that *removes* some water when adjacent to soil; in the kernel for soil (or a soil-moisture channel), include weights that *add* the corresponding amount. This effectively transfers water from one channel to another in neighboring cells. Because conv can sum contributions from different input channels, it's possible to encode such rules in the kernel weights (e.g. weight = +α for neighbor's water in the soil moisture output channel). Alternatively, you can handle absorption with explicit post-convolution logic: find where water and soil co-occur and transfer min(water, capacity) from water's tensor to soil's tensor.

PyTorch Implementation Pattern: Use `torch.nn.functional.conv2d`/`conv3d` with appropriately shaped kernels. For example, a 2D update might use `F.conv2d(state.unsqueeze(0), weight, bias=None, stride=1, padding=1, groups=C)`, where `state` is shape `(1,C,H,W)`. Grouped conv lets each material's update consider only its own channels if desired [9] [7]. You can then apply nonlinear fixes: e.g. `delta = torch.clamp(delta, min=0, max=some_limit)` to ensure no cell is overfilled or negative. The result `delta` is added to the old state (or directly serve as the new state if kernels already compute new absolute values).

Example: In a falling-sand CA, one could implement a *two-phase conv*: First, use conv to compute how much mass would like to move down or sideways (this could output multiple feature maps: e.g. one for "mass leaving cell downward", one for "mass entering from above", etc.). Then enforce constraints: if two neighbors both send sand into the same cell, cap the total to the cell's capacity. In PyTorch, you might compute neighbor contributions with `torch.roll` instead of conv for clarity: e.g. `down = torch.roll(sand, shifts=+1, dims=1)` gives sand shifted down. Then compute flow: `flow_down = (sand - down) * flow_rate * (down_capacity)`, where `down_capacity` could be something like `(1 - down_filled_fraction)` or an indicator that the cell below is emptier. Use `torch.clamp` on `flow_down` to ≥0 (only flow if above > below). Similar terms can be computed for sideways. Finally update: `new_sand = sand - flow_down_out - flow_side_out + flow_down_in + flow_side_in`. All these are tensor operations that PyTorch will execute in parallel on GPU.

Boundary conditions: Handle edges by padding the grid (e.g. use `padding=1` in conv with appropriate padding mode, or manually pad a "border" of zeros or immovable bedrock). A common trick is to pad the tensor by one or more cells so that the conv/neighborhood always has valid neighbors [10]. This avoids costly Python-side boundary checks and branchy code, letting the GPU churn on a consistent grid.

3. Unified Rule Generalization

The goal is one **unified update function** that covers gravity, dispersion, and absorption without separate hand-coded phases. Achieve this by treating all these effects as *neighbor exchanges of material*, differing only in rate or direction bias. In other words, define a **single neighborhood rule** that, for each cell, looks at the current cell and its neighbors' material amounts and computes *all* transfers at once.

A powerful abstraction is to use **learned update rules** or parameterized functions that can generalize across behaviors. For example, you could train a small CNN or use a rule table to output the next state given the 3×3 neighborhood [11]. Google's *Neural CA* research demonstrated that a convolutional "update network" can learn complex self-organizing rules [12]. In our case, you might not train the rule with gradient descent, but you can still **design it like a neural network layer**: use conv filters to gather neighbor info, then non-linear functions (ReLU, min/max, etc.) to decide flows, then maybe another conv to distribute those flows. This multi-step computation is still one *PyTorch graph* executed per tick – thus conceptually a single fused "pass."

Parameterizing per Material: Use learnable or tunable tensors for material properties that factor into the update. For example, define `flow_strength` as a vector of shape `(C,)` that scales how easily each material moves. Then when computing a flow, multiply by the source material's flow_strength. Absorbency between materials can be a matrix `A[c_target, c_source]` that says what fraction of source converts to target if adjacent. These parameters can be applied in the convolution kernels or in post-processing. Keeping them as tensors means you can adjust or even differentiate through them easily (as in differentiable physics engines that learn material properties by gradient descent [13]).

Unifying Gravity vs. Lateral Flow: Gravity is essentially a *directional bias* in the flow rule. You can incorporate this by weighting the conv kernel such that downward neighbor differences are amplified. Alternatively, you can handle gravity outside the conv by adding a constant "gravity field" that pulls certain materials down. For example, add a small downward flow of any free material each step (could be as simple as `velocity_y[channel] += g` parameter, then handled in conv as if neighbor below had less). Many cellular sand simulations use random or biased choices to break symmetry – e.g. if a grain can fall down-left or down-right, choose one randomly or alternate bias to avoid stable vertical stacks. In a parallel GPU approach, you can encode a left/right bias per time-step (e.g. a flag that flips each frame or a random mask) so that large-scale symmetry breaks. This can be done by having two conv kernels for diagonals and selecting one based on step parity, or by adding a random noise tensor to slightly perturb decisions.

Absorption & Reactions: By treating these as additional channels or as part of the local rule, you avoid writing separate loops for them. For instance, soil absorbing water can be a *first-order reaction*: water -> soil moisture at some rate if co-located. Implement this by a simple elementwise operation on the tensors each step (e.g. `soil_moisture += absorb_rate * min(water, capacity)` and `water -= ...`). Since this is pointwise per cell, it's easily done in PyTorch on the whole grid at once. Alternatively, if absorption requires neighbor interaction (water moves into adjacent soil), that can also be done with convolution: e.g. a kernel that detects water above a dry soil cell and moves some water down into it. The key is that **all these updates happen in the same timestep** on the old state, producing the new state. There's no separate "gravity pass" then "absorption pass" – write a single function `update(state)` that computes **new_state = f(state)** applying all effects.

4. Extending to 3D

The 3D case is a natural extension: use `conv3d` for 3D stencils or `torch.roll` in 3 dimensions for neighbor shifts. Gravity now acts along the z-axis (or whichever axis is "vertical"), and lateral dispersion along x/y. The **pattern of implementation remains the same**, but note that the number of neighbors grows (6 face neighbors in a von Neumann neighborhood, or up to 26 in Moore neighborhood). This increases computation per cell but is still highly parallel. PyTorch's cuDNN-backed conv3d will handle this efficiently, though you should be mindful of memory – a large D×H×W volume can be heavy. If needed, use half-precision (`torch.float16`) to reduce memory bandwidth, or sparse data structures if the world is mostly empty (PyTorch has limited native support for sparse conv, but libraries like MinkowskiEngine can help with sparse grids).

Performance Tip: For large 3D worlds, memory access patterns dominate performance. It can help to use **channels-last memory format** (e.g. `tensor = tensor.contiguous(memory_format=torch.channels_last)` for conv3d) if you have many channels, so that the inner dimension (C) is small for coalesced memory access. Since our C (number of materials) is relatively small, NCHW format is usually fine (and is default for PyTorch convolutions [14]).

5. Parallelism and Race Conditions

Because we update the whole grid in one go (synchronously), we avoid race conditions where two threads write to the same cell – each cell's new state is computed from the old state *without interference*. However, when simulating moving "grains" this synchronous update can cause artifacts (e.g. two sand grains landing in the same spot). In classical CA, one solution is an **alternating update scheme**: e.g. partition the grid into a checkerboard of cells and only allow half to move, then the other half next step. A GPU-friendly variant is the **Margolus neighborhood (block cellular automata)** which updates disjoint 2×2 (or 2×2×2 in 3D) blocks each step, with a shift each iteration [8]. This ensures no two cells simultaneously move into the same space, at the cost of a more complex update pattern. In a PyTorch context, you could implement Margolus by slicing the tensor into sub-blocks or using masking to update alternate cells. If your simulation treats materials in a continuous (fractional) way, outright collisions are less problematic (two half-filled cells can merge in one cell). But if you see unstable oscillations, consider these schemes or add some random jitter to the rules to break perfect symmetry [15] [4].

Another consideration is **double buffering**: always read from the old state and write to a new tensor for the next state (typical in functional programming and ML frameworks). This is naturally how PyTorch works – if you do `new_state = f(old_state)` with tensor ops, you're not updating in-place, so no conflict. Avoid in-place ops on the input tensor as it would violate autograd and also mess up neighbor reads in the same pass.

6. PyTorch Coding Patterns and Optimization

Using PyTorch (or similar frameworks) lets us leverage GPU kernels and even autograd for free. Here are some patterns and tips for efficient implementation:

- **Vectorized Neighborhood Access:** Instead of Python loops over cells, use convolution or tensor shifts. For example, `torch.roll(grid, shifts=1, dims=2)` gives you the grid shifted in one direction. You can get all six neighbor faces in 3D with a few roll calls. Similarly, `F.pad` and slicing can give edge neighbors (or use circular padding if wrapping world). This approach turns neighbor lookups into bulk tensor ops.

- **Unified Convolutional Update:** As described, one `F.conv2d/conv3d` call can produce many outputs. You might design a kernel that outputs one channel per material, where each output channel computes the net gain/loss of that material after all neighbor exchanges. This single conv call can replace many manual computations, and runs in C++ CUDA code under the hood. For example, researchers implemented a 2-channel diffusion+MHD physics update as a chain of grouped conv filters in PyTorch (with each field as a channel) and found it feasible and fast [6].

- **Use JIT or `torch.compile`:** Merging many small tensor ops (add, clamp, etc.) into fewer kernels can improve speed. PyTorch 2.0's `torch.compile` or older TorchScript can fuse operations. One paper noted using just-in-time compilation to fuse the forward pass and enabling cuDNN's auto-tuner yielded good performance for stencil updates [16]. You can wrap your update function with `with torch.inference_mode(): ...` if you don't need gradients, and even `torch.backends.cudnn.benchmark = True` to let cuDNN find the best algo for your conv [16].

- **No-Grad Mode:** If you are not doing differentiable learning through the simulation, wrap the simulation loop in `torch.no_grad()` (or use `.detach()` on intermediate results) to avoid storing massive gradients [17]. This saves memory and overhead. If you *do* want differentiation (e.g. to tune parameters via gradient descent), the above operations are all differentiable by PyTorch (even the discrete-like ones, albeit with piecewise-linear behavior due to clamps). Projects like *PyTorchFire* demonstrate real-time calibration of CA parameters via gradient descent, by making the simulation differentiable [13].

- **GPU Memory and Batch:** If simulating one world, batch size = 1 is fine. But you could simulate many smaller worlds in parallel by stacking them in the batch dimension and using the same conv update on all (PyTorch will vectorize over batch). Ensure your tensor is on CUDA (`device='cuda'`) and watch out for memory – a large 3D tensor might approach a few GB if resolution is high. Consider half precision (`grid = grid.half()`) if that's an issue, as many CA rules are robust to low precision.

- **Performance Pitfalls:** The biggest bottleneck is often *memory bandwidth*, not raw FLOPs. A 3D grid with millions of cells means reading/writing millions of values per step. Try to reuse data on-chip:

e.g. if implementing manually with `torch.roll`, you'll end up holding multiple shifted copies of the grid in memory. Convolution is better here as it streams through once. Also, avoid Python-side conditionals per cell; use vectorized torch operations. If you find yourself writing `for` loops, rethink it with tensor math. Lastly, ensure alignment of threads: accessing tensor slices that are not contiguous can hurt – calling `.contiguous()` on your tensor or using standard layout will help.

7. Examples and References

- **CA in PyTorch:** Moritz et al.'s *Cellular* project defines CA rules as 2D conv kernels in PyTorch, achieving heavy parallelism on GPU [11]. This confirms the viability of using conv for game-of-life style physics.
- **Differentiable Physics Engines:** *ΦFlow* (PhiFlow) is a framework that runs fluid and physics simulations on PyTorch and TensorFlow, using conv operators for things like diffusion and pressure solves [18]. Nvidia's Warp is another tool that JIT-compiles physics update kernels for GPU, but one can get far with just PyTorch ops.
- **Granular Media CA:** Devlin & Schuster (2020) explore sand and granular flow with probabilistic CA rules (using Margolus neighborhoods) [19] [20]. They highlight that CA can capture multi-material interactions (solids, liquids, gases) in games like *Noita* via simple local rules. Our approach is in the same spirit, but leveraging GPU parallelism.
- **Block Cellular Automata:** GelamiSalami's GPU Falling Sand demo uses 2×2 block updates to avoid write conflicts [8]. In our PyTorch design, we achieved the same logical effect by reading old state and writing new state (no simultaneous writes). If needed, their 4-phase shifting strategy can be implemented to remove any directional bias [21].
- **HPC Stencil Codes:** Bastian et al. (2024) implemented diffusion and magneto-hydrodynamics (MHD) equations with PyTorch group conv and found performance within a factor of 2–4× of native CUDA for large grids [22]. This suggests our approach (a form of nonlinear stencil update) can be efficient, and it reinforces using features like grouped conv, fused kernels, and no-grad mode [16].

By following these patterns – multi-channel state, convolutional neighborhood processing, unified rule logic, and careful PyTorch optimization – you can build a **GPU-accelerated voxel simulation** that updates all physical effects in one go. This not only runs fast (millions of cell-updates per second are achievable [23] [24]) but is also flexible: new materials or behaviors can be added by adjusting the local rule kernel or network, without changing the overall architecture. The result is a maintainable, extensible simulation engine harnessing the full parallel power of modern GPUs.

Sources:

- ExxactCorp Blog – *Universes That Learn: Cellular Automata Acceleration* [5] [12]
- Moritz's *cellular* GitHub (2025) – conv-based PyTorch CA rules [11]
- Bastian et al. – *Stencil Computations on GPUs with PyTorch* (2024) [7] [16]
- *PyTorchFire: Differentiable Wildfire CA* – GPU CA for fire spread [13]
- Devlin & Schuster – *Probabilistic CA for Granular Media* (2020) [19] [3]
- GelamiSalami – *GPU Falling Sand CA* README (2023) [8] [21]

[1] [2] [3] [19] [20] [2008.06341] Probabilistic Cellular Automata for Granular Media in Video Games
https://ar5iv.labs.arxiv.org/html/2008.06341

[4] [10] [15] Making a falling sand simulator | Hacker News
https://news.ycombinator.com/item?id=31309616

[5] [12] [23] [24] Universes that Learn: Cellular Automata Applications and Acceleration | Exxact Blog
https://www.exxactcorp.com/blog/Deep-Learning/universes-that-learn-cellular-automata-applications-and-acceleration

[6] [7] [9] [14] [16] [17] [22] arxiv.org
https://arxiv.org/pdf/2406.08923

[8] [21] GitHub - GelamiSalami/GPU-Falling-Sand-CA: GPU Falling sand simulation using block cellular automata
https://github.com/GelamiSalami/GPU-Falling-Sand-CA

[11] GitHub - moritztng/cellular: Cellular Automata in PyTorch with Multiplayer Mode in the Browser via WebRTC :video_game:
https://github.com/moritztng/cellular

[13] [2502.18738] PyTorchFire: A GPU-Accelerated Wildfire Simulator with Differentiable Cellular Automata
https://arxiv.org/abs/2502.18738

[18] [PDF] Differentiable Simulations for PyTorch, TensorFlow and Jax - GitHub
https://raw.githubusercontent.com/mlresearch/v235/main/assets/holl24a/holl24a.pdf