Fast Fluid Simulations with Sparse Volumes on the GPU

Article *in* Computer Graphics Forum · May 2018

DOI: 10.1111/cgf.13350

4 authors, including:

Kui wu
Massachusetts Institute of Technology
23 PUBLICATIONS **275** CITATIONS
SEE PROFILE

Cem Yuksel
University of Utah
65 PUBLICATIONS **877** CITATIONS
SEE PROFILE

EUROGRAPHICS 2018 / D. Gutierrez and A. Sheffer
(Guest Editors)

Volume 37 (2018), Number 2

Fast Fluid Simulations with Sparse Volumes on the GPU

Kui Wu [1] Nghia Truong [1] Cem Yuksel [1] and Rama Hoetzlein [2]

[1] University of Utah [2] NVIDIA Corporation

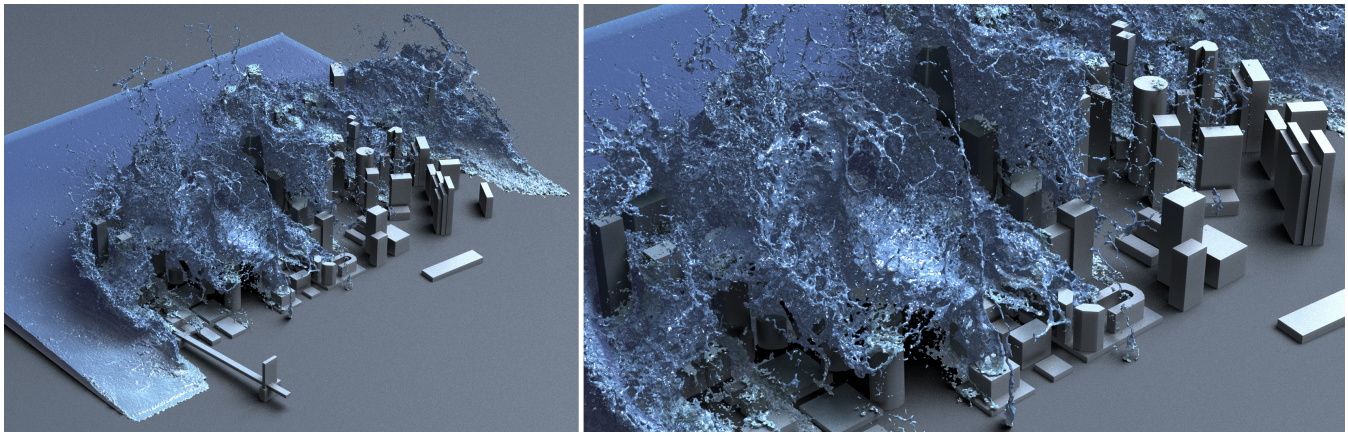

Figure 1: *City scene simulated with 29 million particles on a* $512 \times 256 \times 512$ *grid with our spatially sparse, matrix-free FLIP solver on a Quadro GP100 GPU at an average 1.8 seconds/frame.*

Abstract

We introduce efficient, large scale fluid simulation on GPU hardware using the fluid-implicit particle (FLIP) method over a sparse hierarchy of grids represented in NVIDIA® GVDB Voxels. Our approach handles tens of millions of particles within a virtually unbounded simulation domain. We describe novel techniques for parallel sparse grid hierarchy construction and fast incremental updates on the GPU for moving particles. In addition, our FLIP technique introduces sparse, work efficient parallel data gathering from particle to voxel, and a matrix-free GPU-based conjugate gradient solver optimized for sparse grids. Our results show that our method can achieve up to an order of magnitude faster simulations on the GPU as compared to FLIP simulations running on the CPU.

CCS Concepts

•***Computing methodologies*** → ***Physical simulation;*** *Massively parallel and high-performance simulations;*

1. Introduction

The Fluid-Implicit-Particle (FLIP) method [ZB05] has been popular for simulating various types of fluid phenomena in computer graphics due to its simplicity and low dissipation. As a hybrid technique, high-quality FLIP simulations require both large numbers of particles and high resolution grids. Our method addresses these issues by reducing memory requirements, dynamically tracking the domain, and making use of parallel GPU-based computation residing on a sparse volume representation.

We make use of the GVDB sparse voxel data structure [Hoe16], a GPU-friendly implementation of the VDB sparse grid hierarchy [Mus13]. Computation with sparse grids provides several advantages for fluid simulation. First, it reduces the storage of empty space not occupied by fluid. As we consider simulations entirely in GPU memory this is a crucial factor in running large-scale simulations. Furthermore, with GVDB Voxels the grid size does not have to be pre-defined. As the fluid moves in space new voxels can be allocated automatically as needed. Moreover, voxel data is stored in 3D textures as a collection of dense voxel groups (i.e.

bricks), which allows fast data retrieval during computation and hardware-accelerated trilinear filtering. Finally, neighbor lookups are achieved with apron voxels to accelerate stencil operations on the GPU.

Fluid simulation on sparse grids involves a number of challenges that we address here. First of all, the sparse structure must be rebuilt at every frame from a large number (tens of millions in our tests) of particles. We solve this by introducing an efficient algorithm for rebuilding and updating the tree topology. Second, particle-to-grid rasterization presents performance challenges which we address by introducing a parallel gathering method that efficiently utilizes GPU thread blocks and data coherence while keeping all computation on the GPU. Finally, we introduce a matrix-free conjugate gradient solver for sparse voxel grids to efficiently handle the pressure solver step of FLIP. Together the sparse grids and the matrix-free solver allow us to support much larger volumes than previously achieved in GPU memory as well as reaching up to two orders of magnitude speedup compared to similar dense, multi-core CPU implementations.

Our primary contributions consist of fast dynamic hierarchical topology with incremental update, an efficient GPU-friendly sparse spatial division by subcells, and a fast FLIP simulation with accelerated particle-to-grid rasterization and a matrix-free conjugate gradient solver directly on sparse volumes. Each of these contributions are discussed in detail.

2. Background

There is a large body of work on fluid simulation in computer graphics. Our discussions focus on prior work that is closely related to FLIP and fluid simulations on the GPU. We provide a brief description of the GVDB Voxels structure used for simulation.

2.1. Related Work

Fluid simulation using FLIP [BR86] has been a popular approach since it was introduced to computer graphics [ZB05], along with the particle-in-cell (PIC) method [Har64].

Both PIC and FLIP use particles for advection with the pressure solve performed on the grid. FLIP addresses the numerical diffusion of PIC, but not numerical energy dissipation [MCP*09]. Thus, back and forth error compensation and correction (BFECC) [KLLR07] and the derivative particles [SKK07] methods can be used for reducing numerical dissipation when transferring data between particles and voxels. Researchers later improved the PIC/FLIP method by providing amplified splashing effects [KCC*06, GB13], accurate solid-fluid coupling [BBB07], and proper air-liquid interfaces [BB12]. Recently, Affine Particle-In-Cell (APIC) [JSS*15] were introduced to stably remove the dissipation problems of PIC, providing exact conservation of angular momentum during particle-to-grid transfers. The Integrated Vorticity of Convective Kinematics (IVOCK) [ZBG15] method was introduced for approximately restoring the dissipated vorticity during advection, independent of the advection method.

The PIC/FLIP method is not limited to simulating fluids, as it's first use in computer graphics was in sand simulation [ZB05] and later to solving the internal pressure and frictional stresses in granular materials [NGL10]. PIC/FLIP is also used for handling hair collisions with itself and other objects [MSW*09]. More recently, the Material Point Method (MPM), which generalizes PIC/FLIP using a continuum formulation, was shown to effectively handle simulations of various kinds of materials such as snow [SSC*13], sand [DBD16,KGP*16], viscoelastic fluids, foams and sponges [RGJ*15, YSB*15], and anisotropic elastoplasticity materials [JGT17].

There is also an extensive amount of prior work on accelerating the PIC/FLIP method. The pressure solve on the grid, which involves solving the Poisson equation as a sparse linear system, is often the bottleneck of the PIC/FLIP approach. Decomposing and solving the pressure in parallel over sub-domains has received some attention [NSCL08,WST09,GNS*12]. Other approaches use a low-resolution grid [PTC*10,LZF10,EB14,ATW15] or a multigrid cycle as a preconditioner for a conjugate gradient solver [MST10]. Instead of using uniform grids, tetrahedral discretization [ATW13], far-field grids [ZLC*13], and sparse uniform grids [AGL*17] are used for aggressive adaptivity. Sparse volumetric structures were also used for reducing storage requirements [SABS14]. Recently, Azevedo at el. [ABO16] introduce a topologically correct, boundary-conforming cut-cell mesh to simulate liquid on very coarse grids, while Liu et al. [LMAS16] and Chu et al. [CZY17] present a Schur-complement for solving the Poisson equation on a decomposed domain in parallel. Bailey et al. [BBAW15] also employ sparse volumetric structures on distributed systems for large liquid simulations.

In addition to pressure solve, other methods are introduced to improve performance of the PIC/FLIP method. Lentine et al. [LCPF12] proposed computing an accurate level set representation in order to take large time steps. Rather than using eight particles per voxel as typically in FLIP, Batty and Bridson [BB08] propose using only one particle per voxel with a larger particle radius to transfer particle velocities to the grid using a wider SPH-like kernel. Ferstl et al. [FAW*16] introduce a narrow band FLIP method to only maintain particles within a narrow band of the liquid surface.

The computational power of GPUs have made them an attractive hardware solution for fluid simulation in general. Harris [Har05] implements Eulerian fluid simulation on the GPU with the iterative Jacobi method. Molemaker et al. [MCPN08] introduced Iterated Orthogonal Projections (IOP) as an iterative multigrid-based Poisson solver on the GPU. Horvath and Geiger [HG09] divided pre-simulated fire data on a coarse grid into 2D slices and performed secondary Eulerian Navier-Stokes solvers in parallel across many separate GPUs. Chentanez and Müller developed a specialized multigrid method for pressure projection on the GPU [CM11a,CM11b] and introduced a GPU-friendly sharpening filter that conserves mass locally and globally [CM12]. Recently, Chen et al. [CKIW15] introduced a GPU-based fixed-point method for accelerating Jacobi iterations for real-time painting simulations.

2.2. The GVDB Data Structure

GVDB Voxels is a framework for efficiently representing voxels on a sparse hierarchy of grids [Hoe16] based on the work of [Mus13].

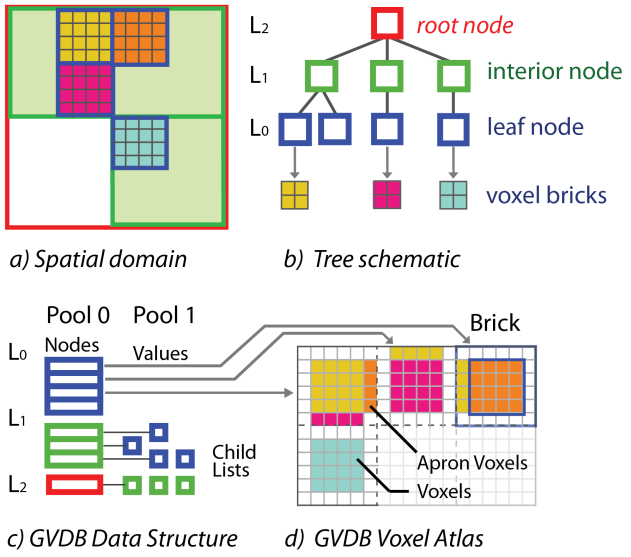

Figure 2: ***The GVDB Data Structure*** *represents sparse voxels covering a spatial domain (a), as a tree of 3D grids (b) whose leaf nodes point to voxel data in bricks. The GVDB data structure makes use of two memory pools (c) to efficiently store basic node attributes in pool 0 and child lists in pool 1. Leaf nodes reference actual voxel data as bricks, which are dense 3D sub-volumes pooled inside a 3D voxel atlas (d) and surrounded by an extra border of apron voxels for later neighbor calculations.*

The GVDB data structure sparsely covers a spatial domain with voxels using a hierarchy of 3D grids as shown in Figure 2a. A configuration vector identifies the resolution of each grid level as a vector of log_2 dimensions. For instance, the $\langle 2,3,4 \rangle$ configuration indicates a tree with three levels, having a root node L_2 with $(2^2)^3 = 64$ children, intermediate nodes at level L_1 with $(2^3)^3 = 512$ children, and leaf nodes at level L_0 referring to dense bricks of $(2^4)^3 = 4,096$ voxels. This configuration can address a volumetric domain of $(2^{2+3+4})^3 \approx 134$ million voxels. GVDB Voxels allows for up to five level trees by default, giving a very large address space. Levels are numbered from zero (leaf nodes) to L (root node) to conveniently allow the tree height to change, as in Figure 2b.

For efficient GPU memory management, GVDB Voxels stores the tree hierarchy with two pools of nodes, shown in Figure 2c. Each node $N(i)_l$ at a given level l is stored in memory pool group $P0_l$ and consists of a world space position, parent index and a child list stored in a separate pool $P1_l$. For interior nodes, each child in the list is a reference to a node in $P0_{l-1}$ at the next lowest level. Leaf nodes at level 0 have no children and instead maintain a reference to a *brick* of voxels stored in a 3D texture atlas.

Whereas the node pools store the GVDB *topology*, the voxel data is represented in dense n^3 bricks allocated from a pool of subvolumes in a *voxel atlas*, shown in Figure 2d. The atlas is implemented as a 3D hardware texture to enable trilinear interpolation and GPU texture cache. Channels, or voxel attributes, are supported with multiple atlases.

3. GVDB Voxels for Simulation

The original GVDB data structure was designed to mimic the VDB approach, where bitmasks are used to compact child lists. In practice, storage of the topology is small (<1 MB) compared to the voxel atlas, and bitmask compaction complicates dynamic insertion and removal of nodes. We modify GVDB Voxels to use explicit child lists for each node. Sparseness is still achieved since these lists only occur in occupied nodes. The new implementation removes the bitmask table and uses a null value in $P1$ to indicate an inactive child node. Dynamic insertion and removal of children are now $O(1)$.

Neighbor calculations are accelerated with *apron voxels* that are maintained as an extra voxel border around each brick, and duplicate the values across brick boundaries. Updating apron voxels is non-trivial since the correct values are voxel neighbors in world space (see Figures 2a and 2d). Since apron update is called repeatedly to maintain data consistency, and is used during the inner loop of our CG solver, performance is critical. The previous technique launches one kernel along each axis, where each thread resolves one apron voxel in atlas space by traversing the GVDB topology to identify its world-space neighbor. We launch a single kernel covering all six sides of a brick, including edges and corners, with one thread per apron voxel performing the traversal. Performance is improved by 5× over the original three-kernel method.

4. Simulation Algorithm Overview

Our fluid simulation technique is presented in Algorithm 1, and follows the Fluid-Implicit Particle (FLIP) method with additions to account for sparse volumes and to enable computation on the GPU. First, the GVDB topology changes dynamically per frame, so we begin with full or incremental rebuild of the tree (Sections 5.1 and 5.2). Previous volume data is cleared on each frame. Second, to accelerate point-to-voxel rasterization on GPU, including level sets

Algorithm 1 Sparse FLIP simulation

```
1:  procedure SparseFLIP( )
2:      P ← initial points
3:      V ← GVDB structure
4:      for each frame do
5:          if first frame then
6:              V_topo ← full rebuild (P)                    ▷ Section 5.1
7:          else
8:              V_topo ← incremental build (P)               ▷ Section 5.2
9:          end if
10:         V ← resize and clear (V_topo)
11:         S ← insert points in subcells (V, P)             ▷ Section 6
12:         V(vel) ← particles-to-voxels (S, P)              ▷ Section 7
13:         V ← update apron (ρ, vel, marker)
14:         V(vel_old) ← V(vel)
15:         V(div) ← divergence (V(vel))
16:         V(ρ) ← CG pressure solve (V, div)                ▷ Section 8
17:         V(vel) ← pressure-to-velocity (V(ρ))
18:         V ← update apron (V(vel))
19:         P ← advance (V(vel), V(vel_old))
20:     end for
21: end procedure
```

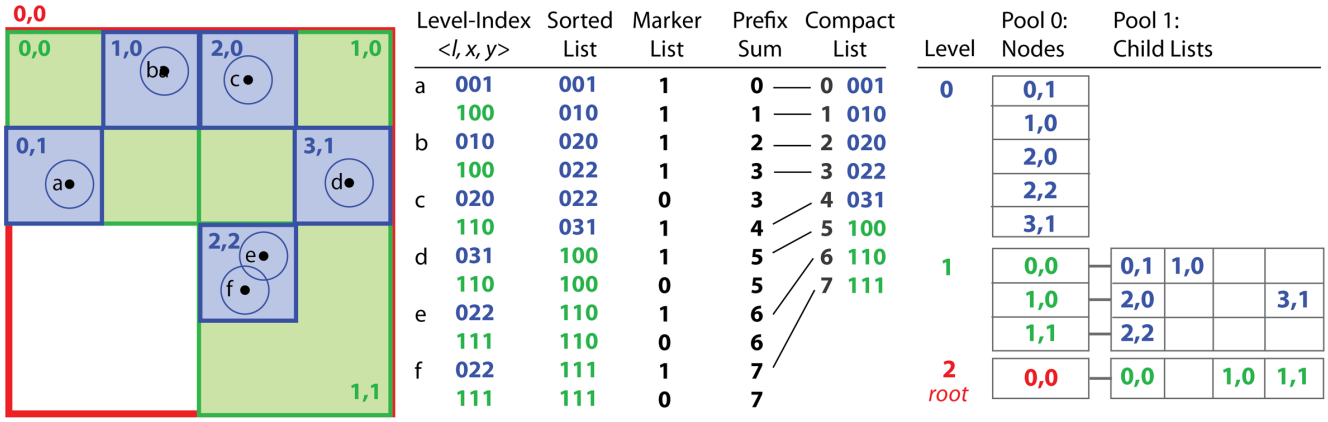

Figure 3: ***Full Topology Rebuild in 2D****: Given input points a, b, c, d, e, and f the goal is to build a containing tree. The index positions of GVDB nodes are at the top-left of the node. After converting points to a list of keys* $\langle \ell, x, y, z \rangle$*, called the level-index list, we using a radix sort to sort all levels simultaneously. Then, each value of the sorted list is checked with the previous one. If they are the same, set a* 0 *value in the marker list, otherwise set a value of* 1*. From the marker list, we compute the prefix sum to provide absolute positions for the output compact list. The sorted list value is written to the compact list at the prefix location only if the marker value is* 1*. The resulting list can be used to directly populate the multi-level tree topology shown at right. Note that empty boxes in the child lists are null values.*

for rendering, we insert points into *subcells* described in Section 6. Third, we handle sparse brick boundaries with apron updates. Finally, our matrix-free conjugate gradient solver for sparse voxel grids is used to efficiently handle the pressure solver step of FLIP (Section 8). All other steps follow a typical FLIP implementation with GPU acceleration, including divergence, pressure-to-velocity, and advection using old and new grid velocities.

5. Dynamic Topology Construction

During simulation each particle must be covered by a sparse voxel brick to contain it. However, each GVDB brick may contain thousands of particles. If we traverse the tree for each particle, even in parallel, there will be a large number of write conflicts and duplicate nodes. Fortunately each GVDB node has a unique absolute spatial index which can be used to classify particles belonging to it. Thus our solution is to sort particles based on the spatial index at *all* levels simultaneously, and then remove duplicates. The remaining set is the list of unique nodes that will cover all particles. Consequently, all GVDB nodes are identified in parallel and reallocated once. We consider both full and incremental rebuild of the GVDB tree hierarchy using this method, with incremental update of moving particles an order of magnitude faster than full rebuild.

5.1. Full Topology Rebuild

The goal of topology rebuild is to efficiently construct the set of interior and leaf nodes of a GVDB tree covering a set of particles. This is performed in two passes to assign voxels up to the particle radius.

5.1.1. First Pass at Particle Centers

The first pass determines the set of tree nodes that cover particle centers. As the fluid domain is virtual unbounded the tree depth may vary, and is recomputed on each frame from the particle bounding box. The set of covering nodes is determined by generating nodes at all levels in parallel and removing duplicates. Finally, nodes are reallocated and parent-child lists are updated. The details of these steps are as follows:

Step 1. Determine the number of tree levels L**.** We begin by computing the bounding box that contains all particles, which determines the level L of the root node required to contain all particles. Thus, only nodes from levels 0 to $L-1$ are identified in the following steps. The bounding box is computed by a parallel GPU reduction algorithm [Har07].

Step 2. Generate the *level-index list*. We define a *level-index list* to consist of the indices of tree nodes that contain each particle at every level of the hierarchy. Let n be the number of particles. The total size of this list is nL and it is generated with a single pass over all particles. The three indices x, y, and z of a node that contains a particle at level ℓ is determined from the particle position. For each level ℓ, where $0 \leq \ell \leq L-1$, a single concatenated integer value *level-index*, $\langle \ell, x, y, z \rangle$, is written to the list. Thus all indexed positions at multiple levels can be sorted together, reducing the number of GPU kernel launches.

Step 3. Compact the level-index list. The level-index list generated in the previous step contains many duplicated values since each node may contain many particles. We first compact this list using a radix sort to group duplicates values. We then mark each element with 1 if it differs from the preceding element, or 0 otherwise, to construct a *marker list*. A parallel prefix sum (scan) [HSO07] over the marker list provides offsets for the final compacted list. Note that the level-index list, marker list, and prefix sum have the same length. The output compacted list is constructed by finding each unique level-index node using the prefix sum for position and only if its marker list value is 1. Each value in the compacted list corresponds to a unique tree node that must be added to the topology to cover all particles. Figure 3 demonstrates construction of the compacted list in 2D.

Step 4. Allocate and initialize nodes. We allocate the two memory pools of GVDB based on the size of the compacted level-index lists. Then, we initialize the node data by assigning the level and position values of each node. All child-node indices are initialized to a null value.

Step 5. Set child node lists. Finally, we set the child node indices in multiple passes. Each pass launches a kernel per level, descending from L_{N-1} to L_0. For each node $N(i)_l$, its parent is found by traversing the tree from the root. The corresponding child node index of the parent node is set at the same time. By induction the top-down order guarantees that each parent has been added to the topology before processing lower levels. Upon completion the GVDB structure contains correct parent and child lists for every node.

5.1.2. Second Pass for Influencing Box

The second pass guarantees that voxels exist to cover up to a unit edge distance for particle-to-grid velocity transfers, or two times the radius to generate level sets (for rendering). We call this bounding box per particle the *influence box*, shown in Figure 4. The first pass generates GVDB nodes which cover particle centers. While it is possible to modify the first pass to extend the level-index list to cover the influence box, this naïve modification would produce a level-index list eight times larger or more. The second pass is used to quickly generate neighboring bricks touched by influencing boxes.

The goal of this second pass is to generate the relatively few voxel bricks that do not yet exist in the tree, do not contain particle centers, but are required due to the influencing box of the particles. For this pass, we modify Step 2 of the first pass to check whether each corner of a particle's influencing box already exists as a GVDB node. If corner nodes are not found, we add these to the list.

Since only missing nodes are added to the list using atomic indices, a much shorter list is generated in comparison to the first pass. As duplicate nodes may still be included, we perform Step 3 again to compact the list. Step 4 and 5 are identical to the first pass.

5.2. Incremental Rebuild

We observe that the tree topology for two consecutive time steps of a FLIP simulation will be very similar even with large time steps since most particles will be moving within the same node, while others may cross nodes. Therefore, we perform incremental updates rather than rebuilding the topology from scratch. Typically, at each step, we only need to allocate a relatively small number of new nodes and voxels and just a small portion of existing nodes are no longer needed. As a result, incremental updates can be performed much more efficiently than complete reconstruction of the topology.

Our incremental rebuild is very similar to the full topology construction algorithm. The main difference is in Step 2 that generates the level-index list. Note that the desired incremental list is the subset of the full node list which only contains new nodes not already in the existing GVDB tree. Thus, for each particle, we check each corner of the particle's influencing box to determine whether there already exists a node at that level. If so, mark the node; otherwise, $\langle \ell, x, y, z \rangle$ is added to the new node list.

Steps 3 through 5 operate similarly. If an existing node is unmarked during Step 2, the node is no longer needed and can be safely removed from the tree. We remove these nodes with an additional step, using multiple passes in a bottom-up order. We remove the unmarked nodes by setting the corresponding child node indices of parent nodes to a null value. Removed nodes and their bricks are dereferenced but not deleted from memory pools, allowing them to be reused for new bricks which are relinked on each frame.

6. Subcell Division

We introduce *subcells*, as shown in Figure 4, which spatially divide each voxel storage brick into multiple, small sub-volumes organized for GPU hardware. Then, we create particle position and velocity lists to store all particles belonging to, or overlapping, each subcell.

The optimal size of a voxel brick depends on the application, with tradeoffs between performance and occupancy [Hoe16]. Whereas large bricks are ideally suited for disk IO and raytracing, the number of voxels-per-brick typically exceeds a GPU thread block, making local computation inefficient. Therefore we retain GVDB Voxel bricks for storage, and introduce *subcells* to create a logical grouping for thread blocks, shown in Figure 4. Subcells organize a given geometry (e.g. points, polygons) for computation with respect to a set of voxels.

The total memory consumed is the union of particles in every subcell including duplicates that overlap other subcells. At one extreme, if a subcell is only one voxel (1^3), each subcell will be exactly the ideal list of particles impacting that voxel. However, the total list length will be much longer than the original particle list. With larger subcells, more unnecessary particles are checked per subcell, but the total list length will be shorter. We use $4^3 = 64$ voxels per subcell as an even multiple of CUDA warps ($32\times$ threads) and to balance search length with memory consumption while also fitting into a single GPU thread block ($512\times$ threads). To improve hardware coherence, instead of storing particle indices, the positions and velocities are copied into each subcell.

Due to duplication in the lists caused by overlaps, the subcell list occupies the most memory during simulation. To further reduce memory usage, we encode position and velocity as 16-bit ushort values dividing by their range. Encoded data only occupy half as much memory overall.

7. Particle-to-Grid Rasterization

A key aspect of a FLIP simulation is to transfer particle velocities to the voxel grid. The weighted sum of nearest particles contribute to any given voxel. Two common approaches are to perform this as either a scatter or a gather. Scatter introduces texture write conflicts as multiple particles write to the same location. It also requires atomic texture ops and produces unbalanced thread workloads. Gathering offers several benefits on the GPU, as found by [KBT*17]. GPU occupancy is improved as each voxel collects values from a fixed shared list of nearby particles, without the need

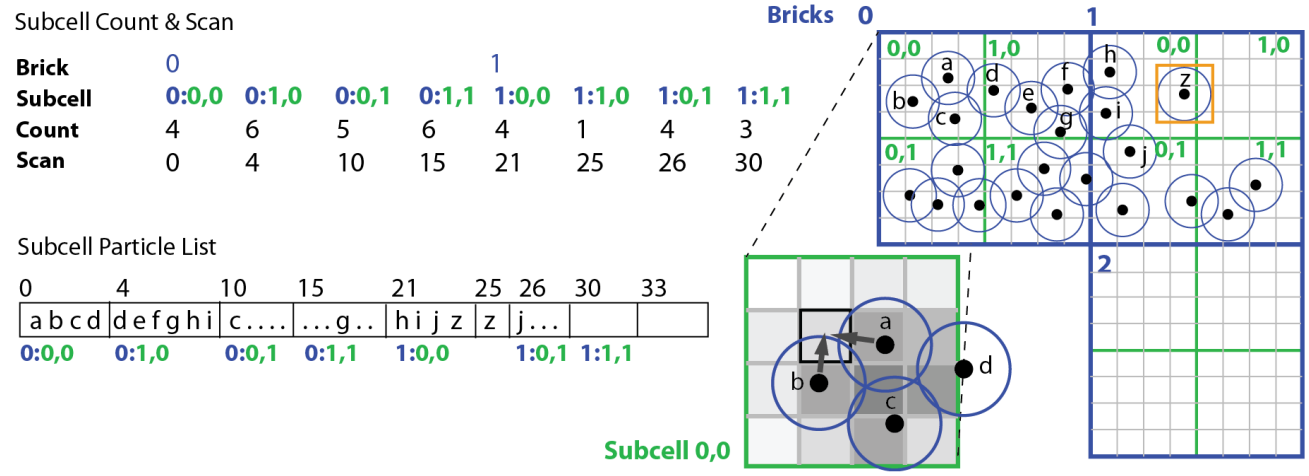

Figure 4: ***Subcell Division****: Subcells are a logical sub-division of voxel bricks into computation units. While bricks organize the sparse domain, and are sized for global performance of raytracing and I/O, they contain too many voxels for efficient GPU compute. Subcells are 64 voxels, sized to fit into a GPU thread block and an even multiple of CUDA warps (32 threads). Every voxel in a subcell shares the same subcell particle list. Notice this requires duplication as, for example, particle d exists in subcells 0,0 and 1,0, but reduces the search length per voxel. During construction, the covered subcells of a particle are determined by rasterizing the particle's influence box (orange).*

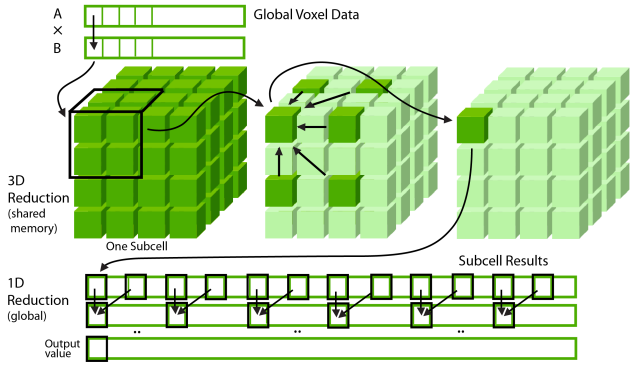

Figure 5: ***Inner Product by 3D Reduction of a volume:*** *To compute inner products for our CG solver, global voxel data from two channels are multiplied into shared memory. Then a 3D reduction by powers-of-two is applied to 4^3 voxels in shared memory to write one value per subcell to a 1D array. Finally, a 1D reduction on the subcell results is performed to return the single output value.*

for atomics. A naïve technique would check the distance of each voxel to every particle, whereas the subcell division described earlier is the ideal shared list of particles for localized, fixed-radius gathering.

A kernel is launched for every active voxel, with each subcell as a GPU grid block. A typical gather accesses each particle once to perform a sum of some attribute. We implement several rasterization kernels to gather velocity and density to generate level set values for rendering. Additionally, many gather operations, such as velocity and fluid-air markers, are combined in a single kernel.

8. Matrix-Free Conjugate Gradient Solver on GPU

The pressure solve is the most important component of an incompressible fluid simulation. Its goal is to solve a linear system $Ax = b$, where x is the unknown pressure values, b is divergence, and A is a sparse matrix that only depends on voxels properties, which can be solid, fluid, or empty. The matrix A is the seven-point Laplacian matrix. Since matrix A is symmetric positive definite, the Conjugate Gradient (CG) algorithm is one of the most commonly used techniques. We introduce a parallel, matrix-free CG solver for FLIP simulation on sparse voxel grids residing directly on a sparse tree hierarchy.

Algorithm 2 provides the pseudo-code for our matrix-free CG solver. Each CG iteration requires one SpMV (Line 10), two inner products (Line 11 and 15), and three vector additions (Line 12, 13, and 17). Vector r, b, q, d, and x are stored as channels in separate 3D GVDB textures. Note that apron voxels in vector d has to be updated before being used in SpMV(Line 10), which will accesses neighbor voxels. The solver termination criteria is checked using the residual norm, read back to the CPU every ten iterations to reduce pipeline stalls (Line 20). Final particle advection is a straight forward sampling of the voxel data at particle locations. Corresponding to the FLIP technique, the old and new grid velocities are sampled and subtracted to give the new particle velocity. Hardware interpolation is supported by GVDB Voxels to retrieve velocity values at arbitrary positions.

The most expensive operation in the CG solver is sparse matrix-vector multiplication (SpMV). The matrix A is a 2D dimensional matrix of size $N \times N$, where N is the total number of active fluid voxels, thus growing rapidly with grid resolution. Even with sparse matrix libraries, the matrix would quickly consume GPU memory for large systems. Therefore, following a matrix-free approach [MGSS13], we avoid storing the matrix A by constructing

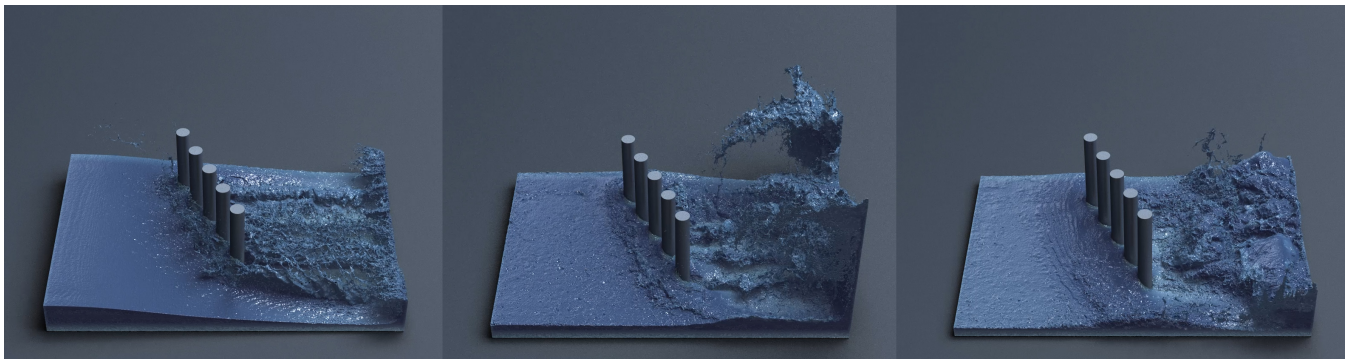

Figure 6: *Column scene with 29 million particles and grid size of* $450 \times 300 \times 300$. *Simulation takes 2.4 seconds per frame with* $1e^{-4}$ *CG solver epsilon on a Quadro GP100. Rendered with NVIDIA® GVDB Voxels and OptiX at twice the grid resolution.*

matrix values from voxels directly as needed on-the-fly. Each row of matrix A corresponds to one pressure equation, in which only seven entries are non-zero at maximum. The diagonal value corresponds to the number of non-solid neighboring voxels and the other six values depend on the six neighboring voxel properties. Due to apron voxels co-located with bricks, immediate neighbors can be accessed as direct texture fetches without tree traversal.

To compute SpMV, one thread per voxel is launched and six neighboring voxel values are checked to compute the inner product between the input vector and the row of matrix A. The result is written to the output vector directly. For instance, to compute the matrix-vector multiplication of A with vector b at row i, we examine the six neighbors of voxel i (x,y,z) to retrieve scalars s_0, s_1, s_2, s_3, s_4, and s_5 on voxels $(x\pm1,y,z)$, $(x,y\pm1,z)$, $(x,y,z\pm1)$. Note that due to GVDB apron voxels, neighbors are already available in GPU texture cache co-located in the same brick as voxel i. For the result vector c, we compute $c_i = b_i \sum_0^5 s_n - \sum_0^5 b_n s_n$. While mapping between world space and atlas space can be performed in constant time with GVDB helper functions, since CG solver operations only require neighbor locality these kernels can be launched directly in atlas space. Result values are written directly to another output channel.

Vector addition is straightforward using a GVDB compute kernel over two channels, but the inner product of two vectors, used for computing the residual in the CG solver, is a complicated operation. For the inner product we perform a three dimensional parallel reduction. Given vector a and b in one subcell, each thread multiplies a_i with b_i and writes to 4^3 shared memory. Then, a 3D reduction is performed in shared memory as shown in Figure 5. Each sub-group of eight voxels will be reduced to one value on each iteration. The 3D reduction step ends with one accumulated value written to a global 2D array. The final step performs a similar 1D array reduction on the results of each subcell to get the final product value.

9. Results

Our results show greatly improved simulation times compared to CPU implementations of FLIP. We perform tests on a variety of scenes at different scales. Performance is measured for topology rebuild, simulation experiments on both GPU and CPU, and memory usage. The CPU performance is measured on a 4-core (8 threads) Intel Core i7 6700K 4.0GHz with 32 GB memory and GPU performance is measured on an NVIDIA Quadro GP100 with 16 GB GPU memory.

Algorithm 2 Matrix-free Conjugate Gradient Solver

▷ *Given the inputs divergence b, starting value x, maximum of iterations* $i_{\max}$, *and error tolerance* ε
procedure MatrixFreeConjugateGradientSolver$(b, x, i_{\max}, \varepsilon)$
  $i = 0$
  $r = b - \mathbf{SpMV}(x)$
  $d = r$
  $\delta_{new} = \mathbf{InnerProduct}(r, r)$
  $\delta_0 = \delta_{new}$
  while $i < i_{\max}$ **do**
    UpdateApron(d)
    $q = \mathbf{SpMV}(d)$
    $\alpha = \delta_{new} / \mathbf{InnerProduct}(d, q)$
    $x = x + \alpha d$
    $r = b - \alpha q$
    $\delta_{old} = \delta_{new}$
    $\delta_{new} = \mathbf{InnerProduct}(r, r)$
    $\beta = \delta_{new} / \delta_{old}$
    $d = r + \beta d$
    $i = i + 1$
    if $i \bmod 10 == 0$ **then**
      if $\delta_{new} > \varepsilon^2 \delta_0$ **then** ▷ *Read back to CPU for check*
        Stop solver and return
      end if
    end if
  end while
end procedure

9.1. Topology Construction

We measure the performance of the topology construction using 4M, 8M, 16M, 32M, and 64M particles with the dam break example, Figure 8. The performances of full CPU, full GPU, and incremental GPU rebuilds are shown in Table 1. Full topology rebuild on GPU is 7–20× faster than CPU rebuild, and incremental rebuild is from 80–180× faster than the original CPU rebuild.

More importantly, incremental topology update requires less than 2% memory usage as compared to full topology build on the

GPU, since the update lists are much smaller. Therefore, in our simulations, a full GPU topology build is only used in the first frame, then the memory is released and made available for later operations, allowing for larger simulations.

Table 1: *Performance of our dynamic topology for full build on CPU, full build on GPU, and incremental on GPU for one frame (in milliseconds). Data is measured with the dam break example.*

Particles #	CPU Full	GPU Full		GPU Incremental	
4M	384	56	(7×)	**4.8**	(80×)
8M	807	96	(8×)	**5.6**	(144×)
16M	1598	204	(8×)	**9.6**	(167×)
32M	3249	313	(10×)	**18.7**	(174×)
65M	6368	366	(17×)	**35.5**	(179×)

9.2. Simulation Performance

We compare the performance of our GPU simulations to three different multi-threaded CPU implementations. The first two use a dense grid with and without an incomplete Cholesky preconditioner for improved performance (Dense CPU CG/PCG). The third comparison is to a CG solver on CPU which achieves sparseness by aggregating active voxels (Sparse CPU CG). Although different from brick-based storage in GVDB, we use this to investigate CG solver performance as the systems will be of similar size.

These three solvers are tested on two relatively simple scenes, see Figures 8 and 9, using different grid sizes and particle counts. As shown in Table 2, our projection step is up to an order of magnitude faster than the CPU-based sparse CG solver. Note that we use 1e-4 as the CG tolerance for all tests. Compared with Dense CPU PCG, which requires one third the iterations, our method is still 10× to 28× faster. Compared with the Sparse CPU CG solver, the performance of which highly depends of the sparsity of grid structure, our GPU solver still achieves a 6× to 10× speedup. In the future we seek to investigate preconditioners suitable for parallel hardware, further improving performance.

Table 3 demonstrates the timing for each step of our simulation. By using our incremental method, the time for topology construction is less than 3% of the total time. The CG solver is still the most expensive part, requiring kernel launches for SpMV, two inner products, three vector additions, and update apron for each iteration. With hardware trilinear interpolation, the final advection step takes around 1% of the total time. We found that performance depends highly on the distribution of particles in space. When water splashes and scatters more bricks are required, increasing the ratio of unused (air) to occupied (fluid) voxels. As long as there is one fluid voxel inside a brick, all voxels in the brick need to be processed. This can be alleviated somewhat by reducing the GVDB brick size at the expense of increased topology rebuild and update time.

9.3. Memory Usage

We measure the peak memory usage for the simulation, which determines the size of simulations that can be handled on the GPU. As described in Section 6, subcell lists occupy the most memory during simulation. The next largest usage of memory is voxel channel

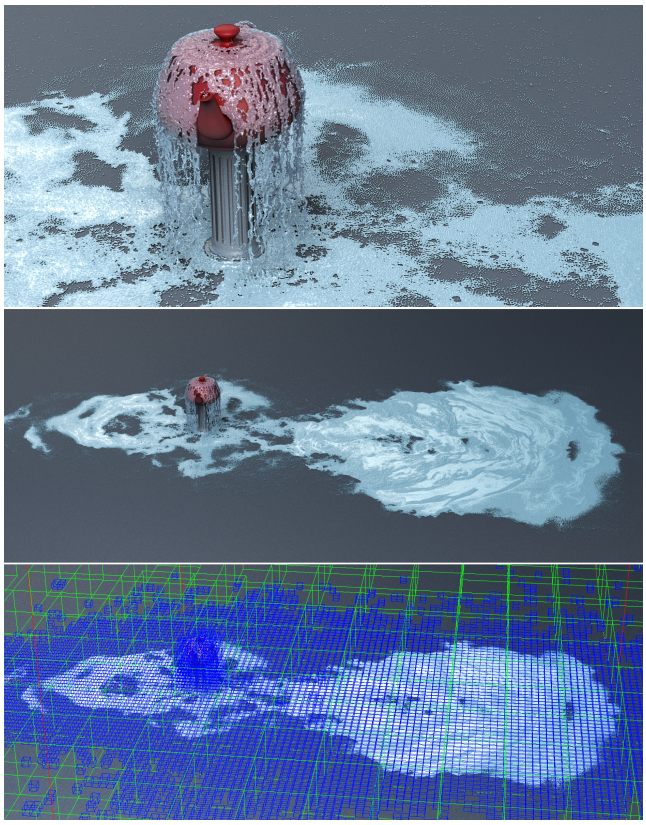

Figure 7: ***Teapot scene****: Simulation of an expanding, sparse domain with 2 million particles. The domain reaches up to* $3360 \times 160 \times 2272$*, with the fluid fully simulated in the sparse bricks indicated in blue. Each brick is* $(2^5)^3$ *= 32768 voxels. Simulation takes 1 second per frame on a Quadro GP100, a lower efficiency than typical for 2M particles due to a higher ratio of unused voxels per brick.*

data. All voxel data is stored in thirteen GVDB channels: density, fluid markers, new and old velocity components at x, y, and z direction, divergence, pressure, and two extra channels for vectors used by the CG solver. Although a single channel is relatively small, around 200 MB for a 20 million particle scene, all 13 channels will take nearly half of the simulation memory. Channel data is stored in 3D textures that dynamically resize to assign bricks covering the fluid.

Full topology build requires a long level-index list for sorting, as well as another list of the same length to store the sorting result, marker list, and prefix sum list. However, the temporary memory can be released after full topology build and used for other purposes such as GVDB channel data. We resize the sorted level-index list, marker list, and prefix sum lists as needed, but do not currently resize the level-index list since it would require an additional pass to compute list length, lowering performance, while it is not the primary memory bottleneck. Note that full topology build requires 2× more memory than incremental topology update in Table 4, but the actually usage of incremental topology update is less than 2% of full rebuild.

Table 2: *Average time per frame for Matrix-Free Conjugate Gradient solver on CPU, CPU with preconditioner, and on GPU.*

			Dense CPU CG			Dense CPU PCG			Sparse CPU CG		Sparse **GPU** CG (GVDB)		
	Particles #	Domain	Iter. #	Iter. Time (ms)	Total (ms)	Iter. #	Iter. Time (ms)	Total (ms)	Iter. Time (ms)	Total (ms)	Iter. Time (ms)	Total (ms)	Speedup
Dam Break	8M	256^3	229	100	22900	49	120	5870	8.3	1901	**0.9**	**206**	**9**×
	27M	384^3	367	334	122578	77	370	28551	27.1	9946	**3.1**	**1138**	**9**×
	65M	512^3	511	792	404712	105	841	88959	69.7	35617	**6.8**	**3475**	**10**×
Water Drop	4M	128^3	243	14	3402	64	21	1392	3.4	826	**0.6**	**146**	**6**×
	13.5M	192^3	350	50	17500	92	80	7444	12.0	4200	**1.7**	**595**	**7**×
	32M	256^3	456	110	50160	111	163	18240	29.1	13270	**3.5**	**1596**	**8**×

Table 3: *Average time per frame for each step in simulation. All times in milliseconds. Our GPU solution sparsely occupies the domain grid, whereas the two CPU solutions cover the domain with voxels. Thus part of our performance gain is due to less work from sparseness. For teapot and flow scene, the domain size is the maximum extent of an unbounded simulation. Each brick is* 32^3.

					GPU time per frame (ms)						
	Particles #	Domain Extents	# Bricks Ave / Peak	CPU Total w/ PCG	Topo. Update	Particle-to-Voxel	CG Solve	Advect	Others	GPU Total per frame (ms)	Speedup
Dam Break	8M	256^3	128 / 207	6892	6	34	206	4	8	**258**	**27**×
	27M	384^3		32050	16	120	1138	14	11	**1299**	**25**×
	65M	512^3		101131	36	298	3475	34	18	**3861**	**26**×
Water Drop	4M	128^3		1858	5	18	146	2	8	**179**	**10**×
	14M	192^3		9067	12	56	595	7	10	**680**	**13**×
	32M	256^3	265 / 512	22080	19	144	1596	17	11	**1787**	**12**×
Column	29M	$450 \times 300 \times 300$	551 / 1006	–	23	138	2260	10	12	**2443**	
City	22M	$512 \times 256 \times 512$	468 / 1033	–	17	112	1721	7	11	**1868**	
Teapot	2M	$3360 \times 360 \times 2272$	800 / 2976	–	20	8	983	2	70	**1083**	
Flow	74M	$1056 \times 288 \times 768$	665 / 812	–	68	270	3377	35	40	**3790**	

9.4. Rendering

Raytracing was performed on a Quadro GP100 using NVIDIA ® OptiX with GVDB Voxels integration at twice the grid resolution of the simulation. We render each scene with one primary, one reflection, two refraction, and one shadow ray using Monte-Carlo raytracing. These five rays constitute one sample with each frame rendered at 128 samples per pixel.

Simulated water is raytraced directly as a sparse volume with GVDB Voxels without conversion to polygons, while collision objects are raytraced as polygonal models. Both use OptiX for hybrid polygon-volume scattering and shading. The particles are gathered as a narrow-band level set for raytracing, which requires extending the influencing box to cover values outside the particle radius. A typical frame is rendered in between 10 to 30 seconds at 1920×1080 resolution with 128 samples/pixel.

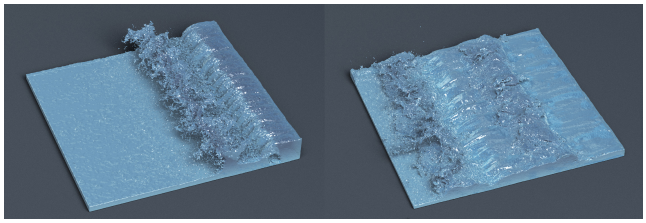

Figure 8: ***Dam break****: Test scene simulated with 8, 27 and 65 million particles at* 256^3, 384^3 *and* 512^3. *Simulation at* 256^3 *shown.*

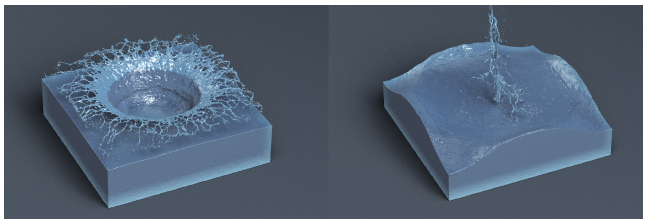

Figure 9: ***Water drop****: Test scene simulated with 4, 14 and 32 million particles at* 128^3, 192^3 *and* 256^3. *Simulation at* 256^3 *shown.*

10. Conclusions and Future Work

We have presented a sparse, efficient, GPU-based FLIP simulation for fluids on virtually unbounded domains. Our method builds a sparse tree topology, incrementally updates the tree, efficiently sorts particles into subcells, and performs the FLIP simulation steps with a fast, matrix-free Conjugate Gradient solver over millions of particles. Overall we can achieve simulation times that are up to an order of magnitude faster than similar CPU-based solutions.

We describe a matrix-free CG solver operating on sparsely placed bricks and optimized for GPU hardware. Rather than storing the sparse matrix A, voxel values are examined directly as needed on-the-fly, greatly reducing memory usage and enabling much larger simulations in GPU memory. Accelerated solver operations include fast 3D reduction for inner products, efficient apron updates for neighbors, and hardware interpolated particle advection. Our system can simulate up to 74 million particles with a few seconds per frame on modern GPU hardware.

Although the performance of our method is already significantly

Table 4: *Peak Memory Usage in megabytes (MB). Object refers to voxel grid for collision geometry.*

	Particles #	GVDB Topology (MB)	GVDB Channels (MB)	Topology Build (MB)	Subcell Lists (MB)	Particle Data (MB)	Object (MB)	Total (MB)
Dam Break	8M	0.06	592	120	305	180	–	1233
	27M		1999	445	1120	664	–	4283
	65M		4664	992	2530	1488	–	9708
Water Drop	4M		592	60	148	88	–	900
	14M		592	206	520	308	–	1640
	32M	0.08	1111	498	1258	746	–	3636
Column	29M	0.09	2221	441	1120	662	56	4583
City	22M	0.11	1851	346	876	518	46	3762
Teapot	2M	0.78	2591	43	70	43	50	3231
Flow	74M	0.14	2517	1127	2818	1691	–	8931

higher than CPU-based solvers, we foresee several potential future improvements. It is possible to introduce a GPU-friendly preconditioner such as a multi-grid technique similar to McAdams et al. [MST10], but implemented on the sparse domain. Another possible improvement would be to implement narrow band FLIP to reduce the number of particles [FAW*16], as particle insertion and sorting into subcells is currently our highest performance cost. Finally, the GVDB topology is naturally suited to an out-of-core approach as the space can be divided along tree boundaries.

Several limitations may be overcome in future work. Collisions are currently handled by voxelization of polygonal obstacle models into a separate GVDB object, which is sampled on a different sparse grid to identify solid boundary voxels. We have plans to support moving rigid objects by updating the transformation matrix (local reference frame) of this collision grid without needing to revoxelize, and by introducing level sets for boundary conditions.

Regarding rendering, we found surface polygonization and smoothing to reduce quality, and use direct volumetric level set raytracing to preserve fine details. In the future we seek to identify and render isolated particles, which are not currently visible. Whereas raytracing uses trilinear gradients for smoothness, stair-stepping artifacts may be visible (see Figure 10) as we do not enforce the free surface Dirichlet boundary condition up to second order accuracy [GFCK02].

To our knowledge we have introduced the first spatially sparse, complete FLIP solver running entirely on the GPU. Every part of the FLIP simulation pipeline has been optimized for parallel execution on GPU hardware using GVDB Voxels for storage, compute and rendering, with significant increases in performance while reducing memory to handle larger simulations.

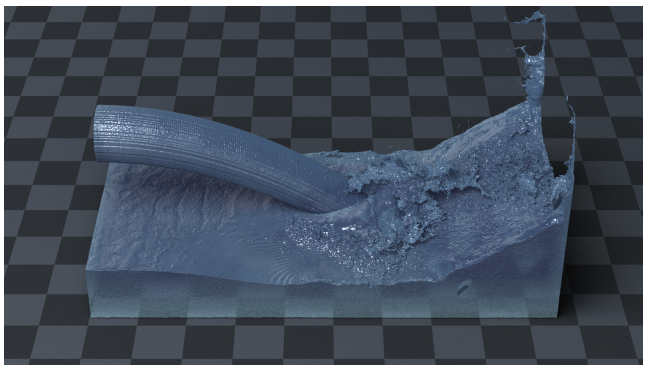

Figure 10: ***Flow****: Simulation of 74M particles in domain* $1056 \times 288 \times 768$

Acknowledgements

The authors would like to specially thank Ken Museth (Weta Digital), Gergely Klar (Dreamworks Animation), and Tristan Lorach (NVIDIA) for their support and guidance. This work was supported in part by NSF grant #1538593.

References

[ABO16] AZEVEDO V. C., BATTY C., OLIVEIRA M. M.: Preserving geometry and topology for fluid flows with thin obstacles and narrow gaps. *ACM Trans. Graph. 35*, 4 (July 2016), 97:1–97:12. 2

[AGL*17] AANJANEYA M., GAO M., LIU H., BATTY C., SIFAKIS E.: Power diagrams and sparse paged grids for high resolution adaptive liquids. *ACM Trans. Graph. 36*, 4 (July 2017), 140:1–140:12. 2

[ATW13] ANDO R., THÜREY N., WOJTAN C.: Highly adaptive liquid simulations on tetrahedral meshes. *ACM Trans. Graph. 32*, 4 (July 2013), 103:1–103:10. 2

[ATW15] ANDO R., THÜREY N., WOJTAN C.: A dimension-reduced pressure solver for liquid simulations. *Comput. Graph. Forum 34*, 2 (May 2015), 473–480. 2

[BB08] BATTY C., BRIDSON R.: Accurate viscous free surfaces for buckling, coiling, and rotating liquids. In *Proceedings of SCA* (2008), SCA '08, Eurographics Association, pp. 219–228. 2

[BB12] BOYD L., BRIDSON R.: Multiflip for energetic two-phase fluid simulation. *ACM Trans. Graph. 31*, 2 (Apr. 2012), 16:1–16:12. 2

[BBAW15] BAILEY D., BIDDLE H., AVRAMOUSSIS N., WARNER M.: Distributing liquids using openvdb. In *ACM SIGGRAPH 2015 Talks* (New York, NY, USA, 2015), SIGGRAPH '15, ACM, pp. 44:1–44:1. 2

[BBB07] BATTY C., BERTAILS F., BRIDSON R.: A fast variational framework for accurate solid-fluid coupling. *ACM Trans. Graph. 26*, 3 (July 2007). 2

[BR86] BRACKBILL J. U., RUPPEL H. M.: Flip: A method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions. *J. Comput. Phys. 65*, 2 (Aug. 1986), 314–343. 2

[CKIW15] CHEN Z., KIM B., ITO D., WANG H.: Wetbrush: GPU-based 3d painting simulation at the bristle level. *ACM Trans. Graph. 34*, 6 (Oct. 2015), 200:1–200:11. 2

[CM11a] CHENTANEZ N., MÜLLER M.: A multigrid fluid pressure solver handling separating solid boundary conditions. In *Proceedings of SCA* (2011), SCA '11, ACM, pp. 83–90. 2

[CM11b] CHENTANEZ N., MÜLLER M.: Real-time eulerian water simulation using a restricted tall cell grid. *ACM Trans. Graph. 30*, 4 (July 2011), 82:1–82:10. 2

[CM12] CHENTANEZ N., MÜLLER M.: Mass-conserving eulerian liquid simulation. In *Proceedings of SCA* (2012), SCA '12, Eurographics Association, pp. 245–254. 2

[CZY17] CHU J., ZAFAR N. B., YANG X.: A schur complement preconditioner for scalable parallel fluid simulation. *ACM Trans. Graph. 36*, 5 (July 2017), 163:1–163:11. 2

[DBD16] DAVIET G., BERTAILS-DESCOUBES F.: A semi-implicit material point method for the continuum simulation of granular materials. *ACM Trans. Graph. 35*, 4 (July 2016), 102:1–102:13. 2

[EB14] EDWARDS E., BRIDSON R.: Detailed water with coarse grids: Combining surface meshes and adaptive discontinuous galerkin. *ACM Trans. Graph. 33*, 4 (July 2014), 136:1–136:9. 2

[FAW*16] FERSTL F., ANDO R., WOJTAN C., WESTERMANN R., THUEREY N.: Narrow band FLIP for liquid simulations. *Computer Graphics Forum (Proc. Eurographics) 35*, 2 (2016), 225–232. 2, 10

[GB13] GERSZEWSKI D., BARGTEIL A. W.: Physics-based animation of large-scale splashing liquids. *ACM Trans. Graph. 32*, 6 (Nov. 2013), 185:1–185:6. 2

[GFCK02] GIBOU F., FEDKIW R. P., CHENG L.-T., KANG M.: A second-order-accurate symmetric discretization of the poisson equation on irregular domains. *J. Comput. Phys. 176*, 1 (Feb. 2002), 205–227. 10

[GNS*12] GOLAS A., NARAIN R., SEWALL J., KRAJCEVSKI P., DUBEY P., LIN M.: Large-scale fluid simulation using velocity-vorticity domain decomposition. *ACM Trans. Graph. 31*, 6 (Nov. 2012), 148:1–148:9. 2

[Har64] HARLOW F. H.: The particle-in-cell computing methods for fluid dynamics. *Methods in Computational Physics 3* (1964), 319–343. 2

[Har05] HARRIS M.: Fast fluid dynamics simulation on the GPU. In *ACM SIGGRAPH 2005 Courses* (New York, NY, USA, 2005), SIGGRAPH '05, ACM. 2

[Har07] HARRIS M.: Optimizing parallel reduction in cuda. *Presentation Packaged with CUDA Toolkit* (2007). 4

[HG09] HORVATH C., GEIGER W.: Directable, high-resolution simulation of fire on the GPU. *ACM Trans. Graph. 28*, 3 (July 2009), 41:1–41:8. 2

[Hoe16] HOETZLEIN R. K.: GVDB: Raytracing sparse voxel database structures on the GPU. In *Proceedings of High Performance Graphics* (2016), HPG '16, Eurographics Association, pp. 109–117. 1, 2, 5

[HSO07] HARRIS M., SENGUPTA S., OWENS J. D.: Parallel Prefix Sum (Scan) with CUDA. In *GPU Gems 3*, Nguyen H., (Ed.). Addison Wesley, August 2007, ch. 39, pp. 851–876. 4

[JGT17] JIANG C., GAST T., TERAN J.: Anisotropic elastoplasticity for cloth, knit and hair frictional contact. *ACM Trans. Graph. 36*, 4 (July 2017), 152:1–152:14. 2

[JSS*15] JIANG C., SCHROEDER C., SELLE A., TERAN J., STOMAKHIN A.: The affine particle-in-cell method. *ACM Trans. Graph. 34*, 4 (July 2015), 51:1–51:10. 2

[KBT*17] KLÁR G., BUDSBERG J., TITUS M., JONES S., MUSETH K.: Production ready mpm simulations. In *ACM SIGGRAPH 2017 Talks* (New York, NY, USA, 2017), SIGGRAPH '17, ACM, pp. 42:1–42:2. 5

[KCC*06] KIM J., CHA D., CHANG B., KOO B., IHM I.: Practical animation of turbulent splashing water. In *Proceedings of SCA* (2006), SCA '06, Eurographics Association, pp. 335–344. 2

[KGP*16] KLÁR G., GAST T., PRADHANA A., FU C., SCHROEDER C., JIANG C., TERAN J.: Drucker-prager elastoplasticity for sand animation. *ACM Trans. Graph. 35*, 4 (July 2016), 103:1–103:12. 2

[KLLR07] KIM B., LIU Y., LLAMAS I., ROSSIGNAC J.: Advections with significantly reduced dissipation and diffusion. *IEEE Transactions on Visualization and Computer Graphics 13*, 1 (Jan. 2007), 135–144. 2

[LCPF12] LENTINE M., CONG M., PATKAR S., FEDKIW R.: Simulating free surface flow with very large time steps. In *Proceedings of SCA* (2012), SCA '12, Eurographics Association, pp. 107–116. 2

[LMAS16] LIU H., MITCHELL N., AANJANEYA M., SIFAKIS E.: A scalable schur-complement fluids solver for heterogeneous compute platforms. *ACM Trans. Graph. 35*, 6 (Nov. 2016), 201:1–201:12. 2

[LZF10] LENTINE M., ZHENG W., FEDKIW R.: A novel algorithm for incompressible flow using only a coarse grid projection. *ACM Trans. Graph. 29*, 4 (July 2010), 114:1–114:9. 2

[MCP*09] MULLEN P., CRANE K., PAVLOV D., TONG Y., DESBRUN M.: Energy-preserving integrators for fluid animation. *ACM Trans. Graph. 28*, 3 (July 2009), 38:1–38:8. 2

[MCPN08] MOLEMAKER J., COHEN J. M., PATEL S., NOH J.: Low viscosity flow simulations for animation. In *Proceedings of SCA* (2008), SCA '08, Eurographics Association, pp. 9–18. 2

[MGSS13] MÜLLER E., GUO X., SCHEICHL R., SHI S.: Matrix-free gpu implementation of a preconditioned conjugate gradient solver for anisotropic elliptic pdes. *Comput. Vis. Sci. 16*, 2 (Apr. 2013), 41–58. 6

[MST10] MCADAMS A., SIFAKIS E., TERAN J.: A parallel multigrid poisson solver for fluids simulation on large grids. In *Proceedings of SCA* (2010), SCA '10, Eurographics Association, pp. 65–74. 2, 10

[MSW*09] MCADAMS A., SELLE A., WARD K., SIFAKIS E., TERAN J.: Detail preserving continuum simulation of straight hair. *ACM Trans. Graph. 28*, 3 (July 2009), 62:1–62:6. 2

[Mus13] MUSETH K.: VDB: High-resolution sparse volumes with dynamic topology. *ACM Trans. Graph. 32*, 3 (July 2013), 27:1–27:22. 1, 2

[NGL10] NARAIN R., GOLAS A., LIN M. C.: Free-flowing granular materials with two-way solid coupling. *ACM Trans. Graph. 29*, 6 (Dec. 2010), 173:1–173:10. 2

[NSCL08] NARAIN R., SEWALL J., CARLSON M., LIN M. C.: Fast animation of turbulence using energy transport and procedural synthesis. *ACM Trans. Graph. 27*, 5 (Dec. 2008), 166:1–166:8. 2

[PTC*10] PFAFF T., THUEREY N., COHEN J., TARIQ S., GROSS M.: Scalable fluid simulation using anisotropic turbulence particles. *ACM Trans. Graph. 29*, 6 (Dec. 2010), 174:1–174:8. 2

[RGJ*15] RAM D., GAST T. F., JIANG C., SCHROEDER C., STOMAKHIN A., TERAN J., KAVEHPOUR P.: A material point method for viscoelastic fluids, foams and sponges. In *Symposium on Computer Animation* (2015), ACM, pp. 157–163. 2

[SABS14] SETALURI R., AANJANEYA M., BAUER S., SIFAKIS E.: Spgrid: A sparse paged grid structure applied to adaptive smoke simulation. *ACM Trans. Graph. 33*, 6 (Nov. 2014), 205:1–205:12. 2

[SKK07] SONG O.-Y., KIM D., KO H.-S.: Derivative particles for simulating detailed movements of fluids. *IEEE Transactions on Visualization and Computer Graphics 13*, 4 (July 2007), 711–719. 2

[SSC*13] STOMAKHIN A., SCHROEDER C., CHAI L., TERAN J., SELLE A.: A material point method for snow simulation. *ACM Trans. Graph. 32*, 4 (July 2013), 102:1–102:10. 2

[WST09] WICKE M., STANTON M., TREUILLE A.: Modular bases for fluid dynamics. *ACM Trans. Graph. 28*, 3 (July 2009), 39:1–39:8. 2

[YSB*15] YUE Y., SMITH B., BATTY C., ZHENG C., GRINSPUN E.: Continuum foam: A material point method for shear-dependent flows. *ACM Trans. Graph. 34*, 5 (Nov. 2015), 160:1–160:20. 2

[ZB05] ZHU Y., BRIDSON R.: Animating sand as a fluid. *ACM Trans. Graph. 24*, 3 (July 2005), 965–972. 1, 2

[ZBG15] ZHANG X., BRIDSON R., GREIF C.: Restoring the missing vorticity in advection-projection fluid solvers. *ACM Trans. Graph. 34*, 4 (July 2015), 52:1–52:8. 2

[ZLC*13] ZHU B., LU W., CONG M., KIM B., FEDKIW R.: A new grid structure for domain extension. *ACM Trans. Graph. 32*, 4 (July 2013), 63:1–63:12. 2