Memory-Constrained Fluid Simulation on the GPU

by

Wouter Raateland

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday, August 30, 2021 at 2:30 PM.

Student number:	4384938	
Project duration:	November 9, 2020 – August 30, 2021	
Thesis committee:	Prof. Dr. Elmar Eisemann,	EEMCS, TU Delft
	Dr. Klaus Hildebrandt,	EEMCS, TU Delft
	Dr. Cynthia Liem,	EEMCS, TU Delft

An electronic version of this thesis is available at `https://repository.tudelft.nl`.

TU Delft

Contents

Article **1**

1 Introduction **2**

2 Background **3**
2.1 Related Work . . . 3
2.2 SPGrid . . . 4

3 Data-Structure **4**
3.1 Random access . . . 5
3.2 Stencil access . . . 6
3.3 Restriction and Prolongation . . . 6
3.4 Incompressible Fluid Solver . . . 7

4 Topology Adaptation **7**
4.1 Basic operations . . . 8
4.2 Adaptation setup . . . 9
4.3 Re-arrangement . . . 9
4.4 Accuracy . . . 10

5 Boundary Conditions **10**
5.1 Approximating Fluidity . . . 11
5.2 Handling partially solid cells . . . 12
5.2.1 Advection . . . 12
5.2.2 Diffusion . . . 12
5.3 Performance . . . 13

6 Terrain-Atmosphere Interaction **14**
6.1 Three-Layer System . . . 14
6.2 Expressiveness of our approach . . . 15

7 Results **15**
7.1 Comparison to a uniform grid . . . 15
7.2 Comparison to SPGrid . . . 16
7.3 Comparison to GVDB . . . 16

8 Conclusions and Future Work **17**

A Quantities per Simulation **21**

Supplementary Material **22**

1 Fluid Simulations **22**
1.1 Navier-Stokes Equations for Incompressible Flow . . . 22
1.2 Semi-Lagrangian Advection . . . 23
1.3 Solving the Pressure Poisson Equation . . . 23

2 Alternative Adaptive Grid Structures **25**
2.1 GVDB . . . 25
2.2 Adaptive Staggered-Tilted Grid . . . 25
2.3 OpenFOAM . . . 26

3 Signed Distance Functions **26**

4 GPU Programming with CUDA **26**

Note: If you are not familiar with some of the subjects discussed in this thesis, I advise you to read the supplementary material before reading the article.

Memory-Constrained Fluid Simulation on the GPU

Wouter Raateland, Delft University of Technology

August 23, 2021

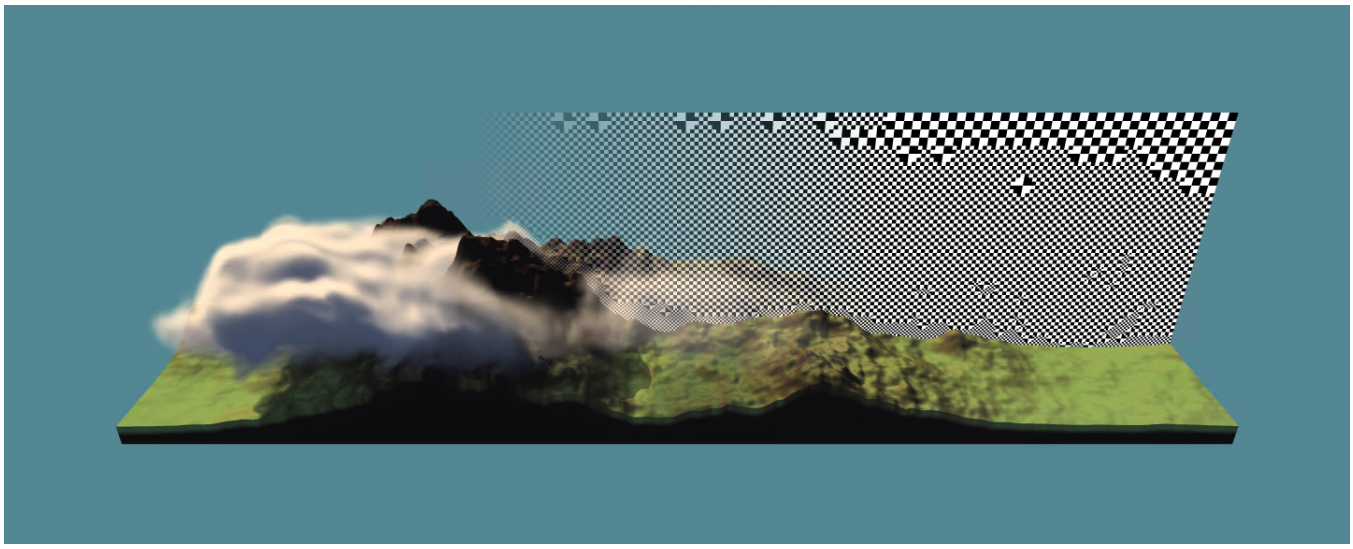


Figure 1: Clouds flowing around a mountain. The checkerboard pattern shows the automatically adapting grid resolution.

Abstract

Grid-based fluid simulations are often limited in resolution by their high memory usage and computational costs. One approach to reducing memory usage and computational costs is to vary the grid resolution over the spatial domain. We introduce DCGrid, a new data structure for fluid simulations. DCGrid is suited for instantiation in GPU memory and allows for varying resolution over the spatial domain. We developed an efficient, optimization-based method for local mesh refinement that automatically adapts the grid resolution according to user-defined parameters during simulations. Additionally, we complement our data structure with an efficient scheme for approximate handling of collisions between fluid and static solids on cells with varying resolutions. We integrate DCGrid in a cloud simulation and extend a terrain-atmosphere interaction model to work with cells of varying resolution and rapidly changing conditions. Furthermore, we demonstrate the performance of our new methods on both simple simulations of smoke flow and complex simulations of weather phenomena and compare them to similar fluid simulations on state-of-the-art adaptive grid structures.

1 Introduction

Grid-based fluid solvers are widely used in generating visual effects and in performing physical simulations. When compared to particle-based, or general mesh-based fluid solvers, grid-based fluid solvers distinguish themselves in their regular structure, the efficiency of random access on them, the ease of performing stencil operations on them, and their suitability for parallelization. One disadvantage of grid-based fluid solvers is that their computational cost and memory usage quickly increase as their resolution increases.

The motivation behind this work is a recent improvement in simulating weather phenomena. Previous works on weather simulation either focused on a narrow range of phenomena or on a multitude of specialized techniques, each capable of modeling a specific phenomenon. Hädrich et al. 2020 instead developed a general grid-based fluid solver capable of simulating a wide variety of complex meteorological phenomena, ranging from low-hanging fog to cumulus clouds. As they rely on uniform grids and are focused on real-time simulations on personal hardware, the resolution on which they can perform simulations is limited. Consequently, the expressive range of their model is also limited.

One approach to reducing the computational cost and memory usage of grid-based fluid simulations is spatial adaptivity. Spatially adaptive grid structures enable higher effective resolutions using fewer cells in total. They do so by allowing for varying resolution throughout the spatial domain. In this work, we introduce a new spatially adaptive grid structure. Our structure is named Dynamic Constrained Grid (DCGrid). DCGrid builds upon another spatially adaptive data structure named Sparse Paged Grid (SPGrid) (Setaluri et al. 2014). We introduce a method for automatically adapting grid topology while respecting memory constraints. We focus on a high-performance GPU implementation of a fluid solver on DCGrid.

Furthermore, we integrate DCGrid in the atmospheric model from Hädrich et al. 2020, and we expand this model with a more expressive method for terrain-atmosphere interaction that enables the simulation of a broader range of weather phenomena.

Specifically, the main contributions of this work are:

- Memory-efficient grid structure based on a hierarchy of sparsely populated uniform grids, highly suitable for fluid simulations.
- Efficient, optimization-based, local topology adaptation method that respects memory constraints.
- Approximate solid-fluid interaction model that does not enforce high-resolution cells near boundaries.
- Terrain-atmosphere interaction scheme suited for cells of varying resolution, capable of expressing a wider variety of meteorological phenomena than previous methods.
- CUDA implementation of an incompressible fluid simulation on DCGrid. Our fluid simulation performs multiple times faster than similar fluid simulations on previous cutting-edge adaptive grid structures. It also uses less memory for an equal number of cells.

2 Background

The work presented in this paper builds upon recent contributions from multiple domains. While we cannot exhaustively discuss the total body of work, we review some closely related work on physics-based modeling of clouds and other meteorological phenomena, computational fluid dynamics, and spatially adaptive grid structures. We give a brief overview of the SPGrid structure that provides the basis for DCGrid.

2.1 Related Work

Physics-based simulation of meteorological phenomena has a long history. Kajiya and Von Herzen 1984 introduced one of the first methods for computationally generating and rendering cloud animations. Many refined representations have been presented, to cope with the high computational complexity of cloud simulations. These representations range from geometric- and particle-based (Bouthors and Neyret 2004; Gardner 1985; Neyret 1997), to position-based dynamics (Ferreira Barbosa, Dobashi, and Yamamoto 2015) and layer-based approaches (Vimont et al. 2020). Recently, Hädrich et al. 2020 introduced a method for modeling a wide range of cloud types using a generic fluid solver. For a more thorough overview of methods for simulating clouds, see Vimont et al. 2020.

Stam 1999 introduced an unconditionally stable method for solving velocity advection. This work has been the basis for grid-based fluid solvers for phenomena such as smoke (Ronald Fedkiw, Stam, and Jensen 2001) and water (Foster and Ronald Fedkiw 2001). Over the years, many works have proposed different methods to make fluid dynamics more performant. Notable ones include, Smoothed Particle Hydrodynamics (SPH) (Cornelis et al. 2014; Gissler et al. 2019), hybrid grid-particle approaches such as Fluid Implicit Particle (FLIP) (Nielsen and Bridson 2016; Wu et al. 2018), FLIP on an optimized narrow band (Ferstl et al. 2016), a combination of FLIP and Point in Cell (PIC) for simulating viscous fluids (Zhu and Bridson 2005) and elastic solids (Brandt et al. 2019), and two-way coupling with the Material Point Method (MPM) (Hu et al. 2018). Other works have focused on accurate simulation of certain features of fluid flow, such as realistic vorticity (Zhang, Bridson, and Greif 2015), mass and momentum conservation (Lentine, Aanjaneya, and Ronald Fedkiw 2011), detail preserving advection (Zehnder, Narain, and Thomaszewski 2018), and realistic viscosity using a multigrid method (Aanjaneya, Han, et al. 2019). Recently, Ummenhofer et al. 2019 introduces an efficient Lagrangian fluid simulation using convolutions. Furthermore, multiple works have proposed improvements to the visual quality of fluid simulations, either by generating high resolution features on a low resolution guiding grid (Inglis et al. 2017; Schoentgen et al. 2020; Xie et al. 2018), or using style transfer (Kim et al. 2019; Sato et al. 2018).

The structured nature of uniform grids offers benefits when used in fluid simulations, such as fast stencil access and simple memory management. On the other hand, their uniform nature dictates a uniform distribution of computational resources over the fluid domain. This uniform distribution is often an inaccurate representation of the distribution of interesting features in a fluid domain (e.g., the surface of a liquid, areas close to a boundary, areas with high vorticity). To simulate features of high interest in the most accurate way possible, we would like to focus computational resources on these features. Multiple works have presented different alternations to the uniform grid that achieve this. Losasso, Gibou, and Ron Fedkiw 2004 was the first to introduce a fluid simulation on an octree structure. Museth 2013 introduced OpenVDB, a sparse data structure organized as a tree with a high branching factor. OpenVDB is especially suited for simulating values at a uniform resolution. A GPU implementation of OpenVDB, GVDB has been developed by Hoetzlein 2016. Wu et al. 2018 developed a FLIP simulation on GVDB, capable of simulating and rendering scenes with tens of millions of particles, where the scene topology automatically adapts to the FLIP particles. Setaluri et al. 2014 introduced SPGrid, a spatially adaptive data structure based upon a hierarchy of sparsely populated uniform grids. SPGrid has been used in multiple large-scale fluid simulations (Aanjaneya, Gao, et al. 2017; Liu et al. 2016) and also in an MPM context (Gao et al. 2019; Hu et al. 2018). Recently, Xiao et al. 2020 has designed an alternative approach to adaptivity based on an Adaptive Staggered-Tilted (AST) grid. This method augments a primary grid with a secondary, overlapping grid with tilted cells (i.e., rotated by 45° in 2D). By scaling each tilted cell individually, they achieve fine-grained adaptation on a uniform grid. The secondary grid typically imposes a runtime penalty

of only a few percent. Nielsen, Stamatelos, et al. 2020 proposed an automatic optimization-based grid refinement algorithm for smoke simulations in Bifrost. This algorithm runs in $\mathcal{O}(n)$ time, where n is the number of allocated grid cells. The widely used application OpenFOAM includes an Adaptive Mesh Refinement (AMR) module. The module is not yet optimization-based. Nevertheless, it has enabled simulations with improved accuracy using less computational resources (Cooke et al. 2014; Lapointe et al. 2020).

2.2 SPGrid

SPGrid uses a hierarchy of sparsely populated uniform grids to model a grid with cells of varying resolution. It supports voxels at each hierarchy level, allowing for a natural level of detail (LOD). SPGrid allocates all grids in the hierarchy in virtual memory. It achieves adaptivity by only allocating the active parts of the grids in physical memory. To do so, SPGrid relies on specific properties of Haswell processors.

SPGrid structures each grid in a hierarchy in blocks of cells such that one block occupies exactly one 4KB memory page. This size results in blocks of 4^3 or $4^2 \times 8$ cells, depending on the number of values simulated per cell. SPGrid allocates blocks in the virtual memory span following a Morton encoding for optimal data locality. This structure enables constant time translations between a block's memory address and its location in the grid. These constant time translations allow for very efficient execution of random and stencil access.

To efficiently compute gradients and Laplacians, SPGrid introduces ghost cells. Let cell C_I^l be a cell with index I located at grid l in the hierarchy. Then C_I^l is ghost when:

- C_I^l is not active at level l,
- C_I^l neighbors a cell that is active at level $s \leq l$,
- and there exists a coarse parent of C_I^l at level $l^* > l$ that is active.

Before computing a gradient or a Laplacian, SPGrid upsamples values from coarse parent cells into their corresponding ghost cells in bulk. With these values in place, it then computes gradients and stencils as if the grid were uniform. After performing the computations, values in the ghost cells are copied back into their coarse parent cells. This way, all grid cells are updated without computing values multiple times.

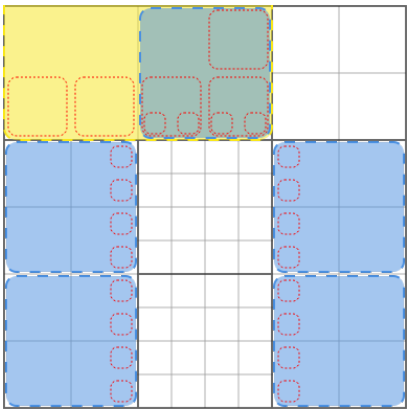


Figure 2: Simple multigrid topology. Ghost cells required to run SPGrid marked in red. Blocks allocated just for ghost cells marked in yellow and blue.

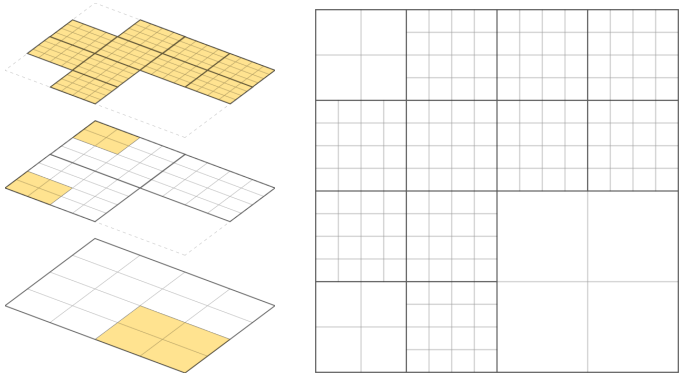
While ghost cells are an effective means for fast computations, they do come with a problem. The problem is that SPGrid allocates cells in blocks. Consequently, allocating only a few ghost cells may require the allocation of multiple complete blocks. This problem is most visible in irregular domains. Figure 2 shows a possible SPGrid hierarchy. In this configuration, SPGrid allocates 6 blocks for active cells. It also allocates 6 extra blocks just for ghost cells.

Adapting topology in a SPGrid hierarchy works by instantiating a new instance with the desired topology and copying values accordingly. This workflow is necessary because some architectures are unable to forget that memory pages have been touched.

3 Data-Structure

Most operations in grid-based fluid simulations make heavy use of random and stencil access. We can implement semi-Lagrangian advection, diffusion, and projection using just random and stencil access. Therefore, our data structure should allow for efficient random and stencil access. The regularity of Cartesian grids allows us to perform many fluid operations in parallel. To run fluid operations as parallel as possible, we run them on the GPU using CUDA kernels. We also designed

DCGrid for allocation in GPU memory. Additionally, fluid simulations often operate on multiple data channels, such as densities, velocities, and auxiliaries. Our structure should support this.



(a) Hierarchy of sparse uniform grids. Cells marked in yellow are active.

(b) Flattened grid hierarchy into one dense multi-resolution grid, showing only active cells.

Figure 3: Two visualisations of the same hierarchy of sparsely populated uniform grids. The thicker lines indicate block boundaries.

We propose DCGrid, a data-structure based on a hierarchy of L sparsely populated uniform grids $G_0, \dots, G_{L-1}$ (figure 3). To achieve good cache coherency, we store cell data in blocks. Each block consists of 2^3 subblocks, which in turn consist of 2^3 cells each. The resolution between two adjacent levels in the grid hierarchy differs by a factor of 2^3. Thus, a block allocated on grid G_l spans the same volume a subblock on grid G_{l+1}. As this scaling factor resembles a mipmap, we call grid G_l mipmap level l. To satisfy memory constraints, we limit the number of blocks $B_{\texttt{max},l}$ that can be allocated simultaneously at each mipmap level $0 \leq l < L$. We also define a global block limit $B_{\texttt{max}} = \sum_{l \leq 0 < L} B_{\texttt{max},l}$.

3.1 Random access

DCGrid allocates blocks of cells in a linear span of memory. Their position in the grid is linked to their index in this linear memory span by a hash table `h`. Let b be a block located at $\mathbf{p}_b$, then the hash of its position is the 32 bit Morton encoding of $\mathbf{p}_b = \lfloor \mathbf{p}/4 \rfloor$. This way of hashing means that a DCGrid instance can have a maximum resolution of $2^{11} \times 2^{11} \times 2^{10}$ blocks, which is equivalent to $8192 \times 8192 \times 4096$ cells. We use a key space of $\texttt{h.size} = 4B_{\texttt{max},l}$ entries for each grid G_l. With this size, we can find the index of the block in the linear memory span I_b in $\mathcal{O}(1)$ time using hash table lookup (algorithm 1).

Algorithm 1: Block index lookup

Input: Position $\mathbf{p}$ and grid G_l. Hash table `h` mapping positions to indices in the linear memory span.
Output: Index I_b, of the block in the linear memory span, or `NOT_FOUND` in case no block exists at position $\mathbf{p}$ in grid G_l.

$k \leftarrow \texttt{hash}(\lfloor \mathbf{p}/4 \rfloor)$
$s \leftarrow k \mod \texttt{h.size}$
$s_0 \leftarrow s$
do
  if $\texttt{h.keys}[s] = k$ **then**
    return $\texttt{h.values}[s]$
  if $\texttt{h.keys}[s] = \texttt{HASH\_EMPTY}$ **then**
    return `NOT_FOUND`
  $s \leftarrow (s + 127) \mod \texttt{h.size}$
while $s \neq s_0$;
return `NOT_FOUND`

DCGrid stores data for each subblock and each cell in separate linear memory spans. These memory spans are ordered equivalently to the memory span used for block data. To illustrate this, let b be the block stored at index I_b. DCGrid stores data for the 8 subblocks contained in b at indices $\{2^3 I_b, 2^3 I_b + 1, \dots, 2^3 I_b + (2^3 - 1)\}$ in the following order:

$$I_s = 2^3 I_b + \texttt{0x100}\left(\left\lfloor \frac{\mathbf{p}.x}{2} \right\rfloor \mod 2\right) + \texttt{0x010}\left(\left\lfloor \frac{\mathbf{p}.y}{2} \right\rfloor \mod 2\right) + \texttt{0x001}\left(\left\lfloor \frac{\mathbf{p}.z}{2} \right\rfloor \mod 2\right) \tag{1}$$

DCGrid stores data for the cells contained in subblock s at indices $\{2^3 I_s, 2^3 I_s + 1, \dots, 2^3 I_s + (2^3 - 1)\}$ in a similar order:

$$I_c = 2^3 I_s + \texttt{0x100}(\mathbf{p}.x \mod 2) + \texttt{0x010}(\mathbf{p}.y \mod 2) + \texttt{0x001}(\mathbf{p}.z \mod 2) \tag{2}$$

In summary, to find the index I_c of cell c located at position $\mathbf{p}$ in grid G_l, we perform three transformations. We first find the index I_b of the block containing c using a hash table lookup. Then, we find the index I_s of the subblock containing c using a simple transformation. Finally, we find the index I_c of cell c using a similar transformation.

3.2 Stencil access

The stencil operations that we focus on are stencils with dimension 3^3. The logical ordering of cells within blocks makes it trivial to compute stencils for cells in the interior of blocks. Computing stencils for cells on the border of blocks, however, requires accessing neighboring blocks. Naively, this would require multiple random access operations and thus multiple hash table lookup operations for each stencil operation.

To more efficiently access adjacent cells in different blocks, we pre-compute an apron of cell indices directly neighboring each block (figure 4). Pre-computing of apron cell indices is performed once, during grid initialization. As the higher resolution mipmap levels will be sparsely occupied, not all blocks will have direct neighbors at the same mipmap level. We could directly include neighbors on other mipmap levels in the apron. Doing so would lead to a different amount of cells neighboring each block. Consequently, we get in non-uniform stencils. Non-uniform stencil computations on a GPU might lead to thread divergence, which reduces their efficiency. Therefore, we want to avoid non-uniform stencils. Instead, we implement an approach that retains the benefit of uniform stencils and works with sparsely occupied grids. To do so, we restrict the layout of the grids in the hierarchy using two rules:

1. For mipmap levels $0 \leq l < L-1$ and for each cell $c \in G_l$, there must exist a parent cell c_p at position $2\lfloor \mathbf{p}/2 \rfloor$ on grid G_{l+1}. We also call c a child cell of c_p.

2. The lowest resolution grid, G_{L-1}, should be densely allocated.

These restrictions ensure that $G_0 \subseteq G_1 \subseteq \cdots \subseteq G_{L-2} \subseteq G_{L-1}$.

To compute the apron cell indices for a block at grid G_l, we iterate over all positions directly adjacent to that block. At each position $\mathbf{p}$, if $\mathbf{p}$ lies inside the domain, we search for a cell in G_l first. If this cell does not exist, we search for a cell at position $\mathbf{p}$ in increasingly coarser mipmap levels. As G_{L-1} covers the complete domain, this process always finds a cell at some level. We now treat all cell indices in aprons as if they originate from the same mipmap level. This process yields a uniform, and thus a performant stencil operation.

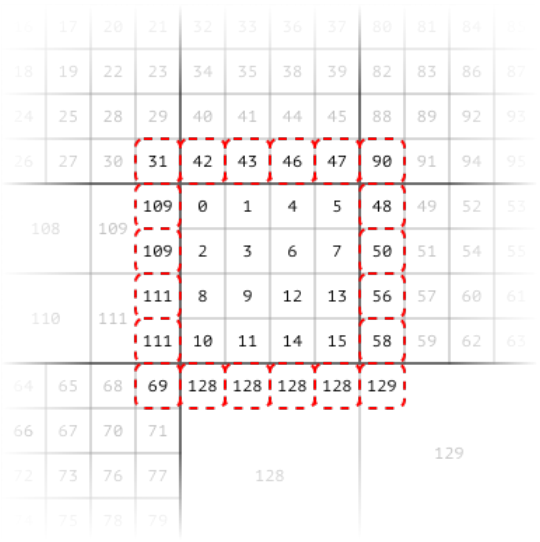
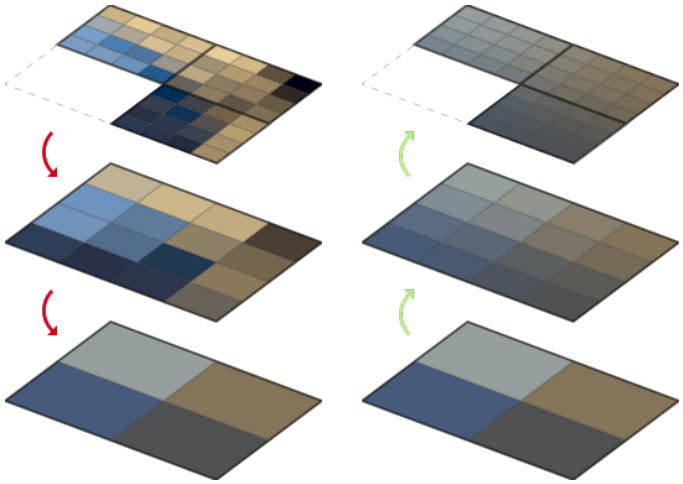


Figure 4: Apron cell indices as calculated for the central block.

3.3 Restriction and Prolongation

By restricting the layout of the grid hierarchy, most of the domain will be covered by multiple cells. Specifically, if cell c covers point $\mathbf{p}$ in the domain, then its parent cell also covers that point. Performing fluid simulation operations on multiple cells covering the same point would be a waste of computational resources. Therefore operations are only performed on the highest resolution cell that covers any point. This cell is called active. Any cell that is the parent cell of another cell is called inactive. To get the correct values in the inactive cells, we perform a simple downsampling routine called restriction (figure 5a). Starting at mipmap level G_0, we average the values of cells sharing the same coarse parent cell. We store this average in the parent cell and repeat on each coarser grid until all cells have values.

The inverse operation of restriction is prolongation (figure 5b). This operation traverses the grid hierarchy in the other direction, from low to high resolution, and transfers values from parent cells into their fine children.



(a) Restriction operation updating cell values by averaging the values of their fine child cells.

(b) Prolongation of restricted values back to high-resolution grid.

Figure 5: Restriction and prolongation operations performed after each other on the same data.

3.4 Incompressible Fluid Solver

Usually, grid-based fluid simulations store velocity and pressure gradients in a staggered MAC grid. Using a staggered MAC arrangement, projection can make the velocity field divergence-free up to a rounding error. The problem with a MAC arrangement is that velocity advection is split into three components and thus takes three times as long. As our work focuses on performance rather than accuracy, we model velocity at cell centers.

Algorithm 2: Multigrid solver

Input: Hierarchy of sparsely populated scalar fields $f_0, \ldots, f_{L-1}$
Output: Hierarchy of sparsely populated scalar fields $u_0, \ldots, u_{L-1}$, approximating the solution to the Poisson equations $\nabla^2 u_l = f_l$ for $0 \leq l < L$.

```
for l = 0 to L - 1 do
    u_l ← 0 // Initialize each value in u_l to 0.
u_{L-1} ← smooth(u_{L-1}, f_{L-1})
for l = L - 2 down to 0 do
    u_l ← prolongate(u_{l+1})
    u_l ← smooth(u_l, f_l)
return u
```

To make the velocity field divergence-free, we use an approximate projection method based on the pressure Laplacian. First, we calculate the divergence on each active cell using a simple 7-point stencil. To compute divergence for all cells in the hierarchy, we apply the restriction operator. Then, we solve the pressure Poisson equation using a multigrid solver (algorithm 2). We typically use a small number of Jacobi iterations as a smoother. Note that our multigrid solver consists of only an upstroke. The restriction operator automatically smooths high-frequency noise out of the divergence field. Therefore, our multigrid solver does not require a downstroke.

We tested our multigrid Poisson solver on a DCGrid instance of smoke flowing around a sphere. Compared to a non-multigrid Poisson solver, our multigrid solver achieves much higher convergence rates using fewer Jacobi iterations (figure 6).

4 Topology Adaptation

We developed an optimization-based local topology adaptation method. Our method distributes computing power and memory usage over the simulation domain by distributing active cells over mipmap levels. In particular, our method distributes active cells according to each cell's priority score p, such that active cells on higher-resolution mipmap levels have higher priority

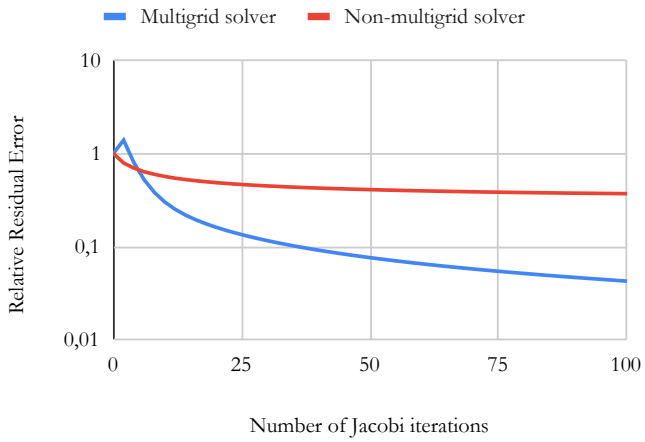


Figure 6: Relative residual error after different numbers of Jacobi iterations our multigrid solver compared to a non-multigrid solver.

scores than active cells on lower-resolution levels. In other words:

$$\begin{aligned} \forall c_1, c_2 : (&\texttt{is\_active}(c_1) \wedge \texttt{is\_active}(c_2) \\ &\wedge \texttt{mipmap\_level}(c_1) < \texttt{mipmap\_evel}(c_2)) \\ &\Rightarrow p(c_1) > p(c_2) \end{aligned} \tag{3}$$

When distributing cells, we need to account for the block limit $B_{\texttt{max},l}$ per mipmap level and the restricted grid layout (section 3.2).

A cell's priority score p can be any user-defined function. Potentially useful priority scores include the velocity gradient, proximity to the camera, or proximity to a boundary. Unless stated otherwise, we use the vorticity magnitude of a cell as its priority score.

4.1 Basic operations

Before introducing our algorithm for topology adaptation, we need to establish the basic operations that constitute the algorithm: subblock refinement and block unrefinement.

To refine subblock s located at mipmap level G_l, we insert a block at mipmap level G_{l-1} spanning the same volume following algorithm 3. The hash table insertion uses CUDA atomic operations to allow for parallel insertions. When a block already exists at position $\mathbf{p}$, or if there is no space left at mipmap level G_{l-1}, we return `NOT_FOUND`. Otherwise, subblock s is marked as inactive, and block $b_{\texttt{refined}}$ is marked as active and positioned in the grid. Finally, the values in block $b_{\texttt{refined}}$ are initialized by prolonging the values from subblock s (section 3.3).

Algorithm 3: Subblock refinement

Input: Subblock s to refine, located at position $\mathbf{p}$ and mipmap level G_l. Hash table `h` mapping positions to indices in the linear memory span.

$b_{\texttt{refined}} \leftarrow \texttt{h.insert}(\mathbf{p}, l-1)$
if $b_{\texttt{refined}} \neq \texttt{NOT\_FOUND}$ **then**
  $\texttt{mark\_inactive}(s)$
  $\texttt{mark\_active}(b_{\texttt{refined}})$
  $b_{\texttt{refined}}.\texttt{position} \leftarrow \mathbf{p}$
  $b_{\texttt{refined}}.\texttt{mipmap\_level} \leftarrow l$
  $b_{\texttt{refined}}.\texttt{values} \leftarrow \texttt{prolongate}(s.\texttt{values})$

To unrefine a block b located at mipmap level G_l, we follow algorithm 4. First, we check if block b is active. If not, then there exists a higher-resolution block at the same position. In this case, we cannot delete b, as that would leave a gap in the grid hierarchy. If b is active, we look up the subblock s that resides at mipmap level $l+1$ at the same position. Because of our restricted grid hierarchy layout, s always exists when $l < L-1$. We mark s as active and delete b from the hash table. Again, this hash table deletion uses CUDA atomic operations and can run in parallel.

A hash table deletion does not clear the previously occupied key in the hash table, leaving the key unusable. After many insertions and deletions, this will slow down lookups in the hash table. Eventually, the hash table will run out of space. To ensure the performance of the hash table, we can refill it periodically.

Algorithm 4: Block unrefinement

Input: Block b to unrefine. Hash table `h` mapping positions to indices in the linear memory span.

```
if is_active(b) then
    s ← h.lookup_subblock(b.position, b.mipmap_level + 1)
    mark_active(s)
    mark_inactive(b)
    h.delete(b)
```

4.2 Adaptation setup

Each DCGrid instance includes at least one mipmap level for which the block limit is such that all its cells can be allocated. In particular, let $G_0, \ldots, G_{L-1}$ be a DCGrid instance with L mipmap levels, then there is a mipmap level l, so that for each $l \leq l' < L$, block limit $B_{\texttt{max},l'}$ is such that $G_{l'}$ can be densely allocated. When initializing a DCGrid instance, we immediately allocate these mipmap levels densely.

Cells on the other mipmap levels $(G_0, \ldots, G_{l-1})$ are allocated iteratively by refining subblocks from $G_1, \ldots, G_l$ respectively. Let G_l, $1 \leq l < L-1$ be a mipmap level on which some blocks are allocated already, i.e., $B_l > 0$. If $B_{l-1} < B_{\texttt{max},l-1}$, then there still space for $N_{l-1} = B_{\texttt{max},l-1} - B_{l-1} > 0$ blocks on mipmap level G_{l-1}. We can now allocate $R_l = \min\{N_{l-1}, 2^3 B_l\}$ blocks on G_{l-1} be refining R_l subblocks from G_l. To select subblocks from G_l, we perform this refinement step only once after each timestep. We first calculate the priority score of each subblock on G_l using a simple CUDA kernel that averages the priority scores of each cell in the subblock. We select the R_l subblocks with the highest priority scores for refinement. Selecting these subblocks is an instance of the k-selection problem. For this problem, a $\mathcal{O}(n)$ algorithms exists (Blum et al. 1973). We solve this problem using the Quick Select algorithm, as this has expected runtime $\mathcal{O}(n)$ and is often faster in practice.

4.3 Re-arrangement

Algorithm 5: Block re-arrangement

Input: Grids $G_0, \ldots, G_{L-1}$, move limit per grid m_l, scalars α and β determining the change in move limits.

```
for l = 0 ... L − 2 do
    ∀s ∈ G_{l+1}, p_s ← calc_subblock_score(s)
    ∀b ∈ G_l, p_b ← calc_block_score(b)
    B_unrefine ← {b ∈ G_l | is_active(b)}
    S_refine ← {s ∈ G_{l+1} | is_active(s)}
    partial_sort_ascending(B_un_refine, m_l)
    partial_sort_descending(S_refine, m_l)
    v_l ← 0
    for (b, s) ∈ zip(B_unrefine, S_refine) do
        if p_b < p_s then
            unrefine(b)
            refine(s)
            v_l ← v_l + 1
    m_l ← max{αm_l, βv_l}
```

In a typical fluid simulation, the distribution of values throughout the domain changes over time. Hence, the parts of the domain that deserve the most attention also change over time. To account for these changes, we allow for re-arrangement of blocks via algorithm 5.

We determine which blocks should be rearranged per mipmap level G_l, starting at the highest resolution mipmap level G_0. First, we calculate the priority scores of each block in G_l and each subblock in G_{l+1}. We then find the set of active blocks $B_{\texttt{unrefine}} \subseteq G_l$ and the set of active subblocks $S_{\texttt{refine}} \subseteq G_{l+1}$. We sort $B_{\texttt{unrefine}}$ ascending $S_{\texttt{refine}}$ descending by their priority scores, and iterate over each pair in order. For each pair, we check if the priority score of the block is lower than the priority score of the subblock. If this is the case, then this pair violates our adaptation objective (equation 3). To move closer towards fulfilling the adaptation objective, we now unrefine the block and refine the subblock.

Sorting all blocks and subblocks is an expensive operation, especially as the amount of blocks grows. Optimally, we would only sort the v_l objective violating pairs. To reduce the number of blocks and subblocks to sort, introduce a move limit m_l, per mipmap level. Move limit m_l indicates that at most m_l subblocks can be refined and m_l blocks can be unrefined at

mipmap level l in one execution of the re-arrangement algorithm. To find the blocks and subblocks that have to be unrefined and refined, we now only need to find the m_l blocks with the lowest priority scores and the m_l subblocks with the highest priority scores. This only requires partial sorting of $B_{\texttt{unrefine}}$ and $S_{\texttt{refine}}$ up to m_l elements, which can be performed in $O(\max\{|B_{\texttt{unrefine}}|, |S_{\texttt{refine}}|\} \log m_l)$ time.

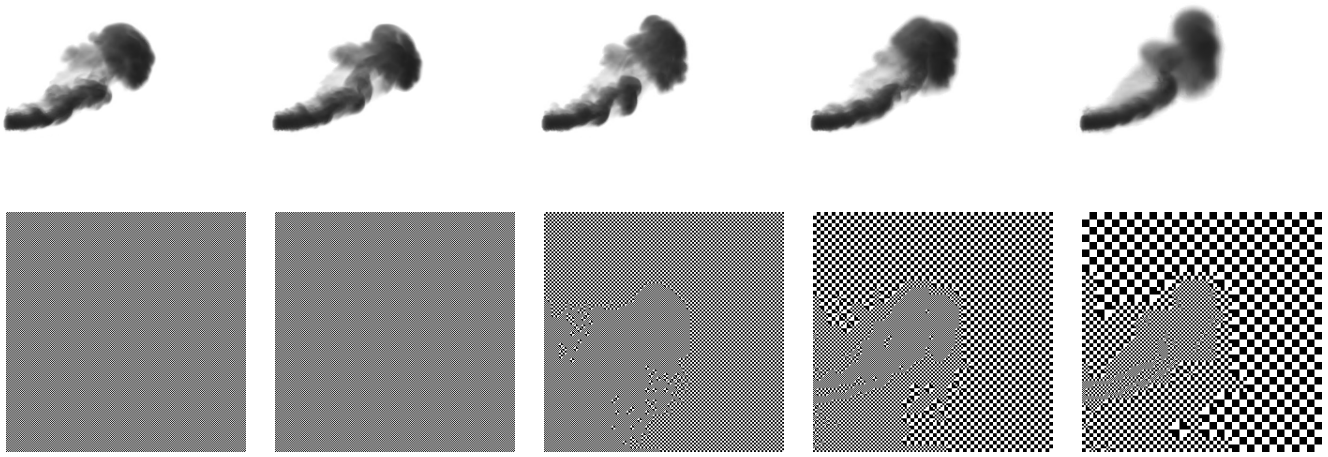
To find reasonable limits m_l, we make three observations:

- If $m_l > v_l$, we sort more blocks and subblocks than necessary, which is a waste of time.
- If $m_l < v_l$, we sort fewer blocks and subblocks than there are objective violating pairs. Doing so leaves some objective violating pairs unchanged, which is undesirable.
- The amount of objective violating pairs v_l in one timestep is often similar to v_l in the next timestep.

From these observations, we conclude that move limit m_l should be larger than the amount of objective violating pairs v_l, but the difference $m_l - v_l$ should as small as possible. We also see that we can predict reasonable limits m_l for the next timestep using the amount of objective violating pairs v_l in the current timestep. In our experiments, we found that using $m_l = \max\{\alpha m_l, \beta v_l\}$, with $\alpha = .8$ and $\beta = 1.5$ gave good results.

After the re-arrangement step, we ensure that all apron cell indices point to the correct cells again. To do so, we calculate the apron cell indices for each newly inserted block and refresh the apron cell indices that pointed to refined or unrefined blocks.

4.4 Accuracy



(a) Uniform, 16.8M cells (b) DCGrid, 19.2M cells (c) DCGrid, 4.8M cells (d) DCGrid, 960K cells (e) DCGrid, 192K cells

Figure 7: Smoke simulation run for 150 timesteps under different memory limits. Effective resolution is 256^3. Each checkerboard square represents one cell.

Figure 7 shows the same simulation run with different limits, and thus with a different number of allocated cells. The high-resolution areas in the grid accurately follow the smoke flow. The simulation with the lowest cell limit (figure 7e) allocates less than 1.5% of the cells used by the simulation on a uniform domain (figure 7a). Still, it faithfully captures the global features of the smoke flow. The highest resolution adaptive simulation (figure 7c) uses one quarter of the cells that the non-adaptive simulations use (figure 7a, 7b). This adaptive simulation reproduces almost all the details visible in the non-adaptive simulations.

5 Boundary Conditions

Other works on fluid simulation on adaptive grids often use high-resolution cells near boundaries and other solids (Aanjaneya, Gao, et al. 2017; Setaluri et al. 2014; Xiao et al. 2020). We use fixed limits on the number of cells that each mipmap level can contain at any time. Always having high-resolution cells near boundaries and other solids would limit the efficacy of the automatic adaptation algorithm. Therefore, we do not enforce cells near boundaries and other solids to have high resolution.

Not enforcing high-resolution cells near boundaries does pose a problem. We now need to handle fluid-solid interactions on cells with different resolutions. As a solution, we propose an approximation scheme. In our scheme, we handle domain boundaries as standard Dirichlet boundary conditions. Inside the domain, however, we do not just mark cells as either solid

or fluid. Instead, we define a fluidity $r_c \in [0,1]$ on each cell c. Let $\Omega \in \mathbb{R}^3$ be the part of the domain that is solid, then Ω^c is the part of the domain that is fluid. Now r_c approximates the part of cell c that overlaps with Ω^c. For example, $r_c = 1$ indicates that cell c is completely fluid and $r_c = 0$ indicates that c is completely solid. Figure 8 illustrates the fluidity of cells at different mipmap levels where Ω is a sphere with radius 4, i.e $\Omega = \{\mathbf{x} \in \mathbb{R}^3 : \|\mathbf{x} - (8,8,8)\| \leq 4\}$.

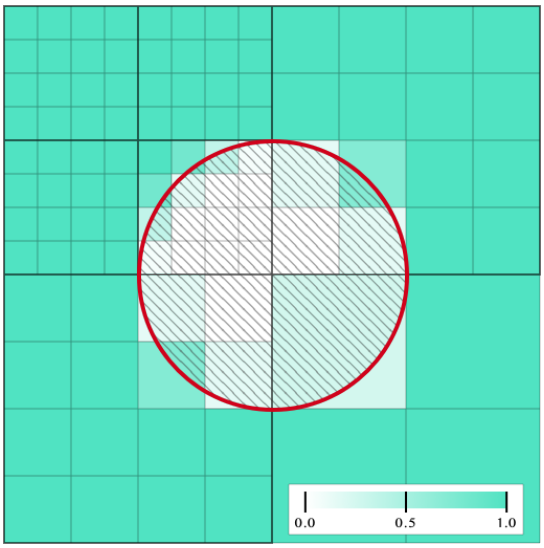


Figure 8: Fluidity r_c for cells of different resolutions against a simple spherical boundary.

5.1 Approximating Fluidity

A naive implementation might regard a cell c at a coarse mipmap level l and position $\mathbf{p} \in \mathbb{R}^3$ as if it were the sum of the high-resolution cells it spans. It would calculate r_c by iterating over all these high-resolution cells, and then taking the average of their fluidity:

$$r_c = \frac{1}{8^l} \sum_{x=\mathbf{p}.x}^{\mathbf{p}.x+2^l} \sum_{y=\mathbf{p}.y}^{\mathbf{p}.y+2^l} \sum_{z=\mathbf{p}.z}^{\mathbf{p}.z+2^l} \mathbf{1}_{\Omega^c}(x,y,z), \tag{4}$$

where $\mathbf{1}_{\Omega^c} : \mathbb{R}^3 \mapsto \{0,1\}$ is the indicator function of Ω^c. This way, approximating the fluidity of coarse cells requires more computations than approximating the fluidity of fine cells. This is contrary to our reason for using coarse cells in the first place: to save computational power.

Our method approximates the fluidity of each cell using a single evaluation of the signed distance function (SDF) of Ω. We define the SDF of Ω, $f_\Omega : \mathbb{R}^3 \mapsto \mathbb{R}$ as usual:

$$f_\Omega(\mathbf{x}) = \begin{cases} d(\mathbf{x}, \partial\Omega) & \text{if} \quad \mathbf{x} \in \Omega^c \\ -d(\mathbf{x}, \partial\Omega) & \text{if} \quad \mathbf{x} \in \Omega, \end{cases} \tag{5}$$

where $\partial\Omega$ denotes the boundary of Ω and

$$d(\mathbf{x}, \partial\Omega) = \inf_{\mathbf{y} \in \partial\Omega} \|\mathbf{x} - \mathbf{y}\|. \tag{6}$$

To use this signed distance function, we use the following property of signed distance functions:

Theorem 1. *Let $\Omega \in \mathbb{R}^3$ be a bounded set with signed distance function $f_\Omega : \mathbb{R}^3 \mapsto \mathbb{R}$ and let $\mathbf{x}, \mathbf{y} \in \mathbb{R}^3$ be two points such that $\mathbf{x} \notin \partial\Omega$ and $\|\mathbf{x} - \mathbf{y}\| < |f_\Omega(\mathbf{x})|$. Then $\mathbf{x}$ and $\mathbf{y}$ are both inside, or both outside of Ω, i.e., $\mathbf{y} \in \Omega$ if and only if $\mathbf{x} \in \Omega$.*

Proof. Let Ω, f_Ω, $\mathbf{x}$ and $\mathbf{y}$ be defined as before. If $\mathbf{x} \in \Omega$ and $\mathbf{y} \in \Omega^c$, or if $\mathbf{x} \in \Omega^c$ and $\mathbf{y} \in \Omega$, then the shortest path between $\mathbf{x}$ and $\mathbf{y}$ crosses $\partial\Omega$ at least once at some point $\mathbf{z} \in \partial\Omega$. It follows that $\|\mathbf{x}-\mathbf{y}\| = \|\mathbf{x}-\mathbf{z}\|+\|\mathbf{y}-\mathbf{z}\| \geq \|\mathbf{x}-\mathbf{z}\| \geq d(\mathbf{x}, \partial\Omega) = |f_\Omega(\mathbf{x})|$. This statement contradicts our condition that $\|\mathbf{x} - \mathbf{y}\| < |f_\Omega(\mathbf{x})|$. Hence, either both $\mathbf{x}, \mathbf{y} \in \Omega$, or both $\mathbf{x}, \mathbf{y} \in \Omega^c$ □

Now, let c be a grid cell at mipmap level l centered around $\mathbf{p} \in \mathbb{R}^3$. We can interpret c as a cube with edge length $s = 2^l$ and thus for each point $\mathbf{p}' \in c$, $\|\mathbf{p} - \mathbf{p}'\| < \frac{1}{2}s\sqrt{3}$.

Using theorem 1, we observe the following:

- If $\mathbf{p} \in \Omega^c$ and $f_\Omega(\mathbf{p}) > \frac{1}{2}s\sqrt{3}$, then the cell is completely fluid, i.e., $c \subseteq \Omega^c$ and hence its fluidity is $r_c = 1$.
- If $\mathbf{p} \in \Omega$ and $f_\Omega(\mathbf{p}) < -\frac{1}{2}s\sqrt{3}$, then the cell is completely solid, i.e., $c \subseteq \Omega$ and hence its fluidity is $r_c = 0$.

- If $-\frac{1}{2}s\sqrt{3} \leq f_\Omega(\mathbf{p}) \leq \frac{1}{2}s\sqrt{3}$, then part of the cell overlaps with Ω and part with Ω^c. In this case, calculating the exact volume of $c \cap \Omega$ would be expensive and thus, we approximate $0 \leq r_c \leq 1$.

We combine these three observations to define a general approximation formula for a cell's fluidity:

$$r_c = \frac{1}{2} + \frac{f_\Omega(\mathbf{p})}{s\sqrt{3}} \tag{7}$$

5.2 Handling partially solid cells

We accounted for partially solid cells in our fluid simulation by modifying some operations. Specifically, we modified the semi-Lagrangian advection and the diffusion operations.

5.2.1 Advection

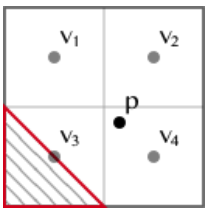


Figure 9: Sampling for advection at point $\mathbf{p}$. Solid areas marked.

To perform semi-Lagrangian advection with partially solid cells, we trace back velocities from cell centers as usual. We differed in the method used for sampling.

Let $\mathbf{p}$ be the point at which to sample for advection, and let c_1, c_2, c_3 and c_4 be the cells surrounding $\mathbf{p}$ centered around $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$ and $\mathbf{p}_4$ respectively (figure 9). Normally, one would sample a value by linearly interpolating values v_1, v_2, v_3 and v_4 depending upon position $\mathbf{p}$. The problem with this way of sampling is that the partially solid cells influence the resulting value too much. To balance the influence of partially solid cells, we implemented a weighted interpolation scheme. Let grid spacing be Δx. For $i \in \{1, 2, 3, 4\}$, we define the weight of cell i as

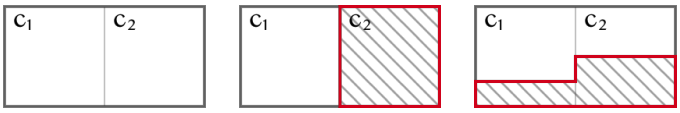
$$w_i = r_{c_i} \frac{(\Delta x - |\mathbf{p}.x - \mathbf{p}_i.x|)(\Delta x - |\mathbf{p}.y - \mathbf{p}_i.y|)(\Delta x - |\mathbf{p}.z - \mathbf{p}_i.z|)}{\Delta x^3} \tag{8}$$

Given values v_i at cell c_i, we interpolate as follows:

$$v = \frac{\sum_{i \in \{1,2,3,4\}} v_i w_i}{\sum_{i \in \{1,2,3,4\}} w_i} \tag{9}$$

When all surrounding cells are completely fluid, i.e., $r_{c_i} = 1$ for $i \in \{1, 2, 3, 4\}$, then this interpolation method is equal to bilinear interpolation. When all surrounding cells are completely solid, i.e., $r_{c_i} = 0$ for $i \in \{1, 2, 3, 4\}$, we return a default value depending on the quantity that is advected.

5.2.2 Diffusion



(a) $r_{c_1} = 1$, $r_{c_2} = 1$ (b) $r_{c_1} = 1$, $r_{c_2} = 0$ (c) $r_{c_1} = \frac{3}{4}$, $r_{c_2} = \frac{1}{2}$

Figure 10: Different scenarios for diffusion. Solid areas marked.

In this section, we consider the diffusion of quantities through the fluid. We implemented the diffusion operation for each cell as a simple seven-point stencil that computes the diffusion between a cell and its 6 direct neighbors. To account for partially solid cells, we adjust the amount of diffusion between a cell and one of its neighbors based on their fluidity.

Let c_1, c_2 be two directly neighboring cells and let α be the amount of diffusion between the cells if both are completely fluid. To calculate the actual amount of diffusion $s(c_1, c_2)$ between the two cells, we consider three scenarios:

- If both cells are completely fluid, diffusion between the cells is not limited (figure 10a).

- If at least one of the cells is completely solid, i.e., $\min\{r_{c_1}, r_{c_2}\} = 0$, there can be no diffusion between them (figure 10b).
- If both cells are partially fluid, i.e., $0 < r_{c_1}, r_{c_2} \leq 1$, then, on average, the area of the diffusion interface is limited by $\min\{r_{c_1}, r_{c_2}\}$. However, also on average, the volume over which diffusion is performed is scaled by $\frac{r_{c_1}+r_{c_2}}{2}$ (figure 10c).

If we combine the three scenarios, we find a general formula for the amount of diffusion between directly neighboring cells c_1 and c_2:

$$s(c_1, c_2) = \alpha \frac{2 \min\{r_{c_1}, r_{c_2}\}}{r_{c_1} + r_{c_2}}. \tag{10}$$

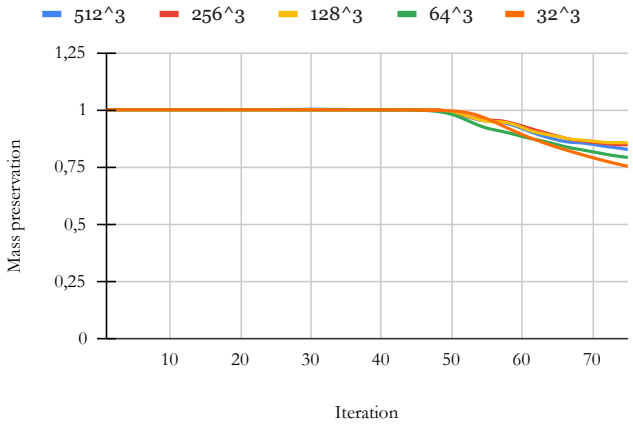


Figure 11: Smoke mass conservation over time for smoke flow around sphere simulated over 75 timesteps on uniform grids with different resolutions. Note that the smoke hit the sphere around timestep 50 on each resolution.

5.3 Performance

(a) $t = 75$, $r = 512^3$, no obstacle (b) $t = 25$, $r = 512^3$ (c) $t = 75$, $r = 512^3$ (d) $t = 75$, $r = 256^3$ (e) $t = 75$, $r = 128^3$ (f) $t = 75$, $r = 64^3$ (g) $t = 75$, $r = 32^3$

Figure 12: Smoke flow around sphere simulated over 75 timesteps using uniform grids with different resolutions.

Figure 12 shows our approximate fluid-solid interactions in action on a simulation of smoke flow past a sphere. We observe similar global behavior when the smoke collides with the spherical solid on vastly different resolutions. Figure 11 reinforces this observation by showing that the mass conservation over time follows a similar trajectory for the higher resolutions 128^3, 256^3, and 512^3. While relatively more mass is absorbed by the solid at lower resolutions 64^3 and 32^3, the global visual behavior still looks plausible.

By pre-computing the fluidity of cells, we can handle arbitrarily complex solids with no runtime overhead and constant memory overhead.

Table 1: Overview of the quantities stored in each cell type in the simulations in this paper.

Cell type	Quantity	Description
Atmosphere	t_f	Potential temperature
	v_f	Vapor density
	c_f	Warm cloud density
	i_f	Ice cloud density
	a_f	Ash density
	r_f	Precipitated rain density
	s_f	Precipitated snow density
	g_f	Precipitated graupel density
Surface	t_s	Temperature
	h_s	Humidity
	f_s	Fuel density (e.g. grass, litter)
	a_s	Ash density
	w_s	Water
	s_s	Snow
	g_s	Graupel
Ground	t_g	Temperature
	h_g	Humidity
	-	Soil Properties (e.g. albedo, heat capacity)

6 Terrain-Atmosphere Interaction

In their recent work, Hädrich et al. 2020 were able to simulate a wide variety of cloud types using a single general model. They relied on first-principle formulations of atmospheric physics and a small set of parameters describing heat and humidity on the ground to simulate anything from mist to cumulus clouds. They modeled the ground as an inlet boundary condition, which worked well when quantities are relatively constant. When quantities at the ground rapidly change, this model could not capture their effects truthfully (figure 15a). We introduce a new method that does capture both relatively constant and rapidly changing quantities at the ground. Our method allows for the simulation of an even wider variety of atmospheric phenomena compared to previous work.

6.1 Three-Layer System

Algorithm 6: Terrain-Atmosphere Interaction

Input: Rectangular domain $D = \{x_0, \dots, x_1\} \times \{y_0, \dots, y_1\} \times \{z_0, \dots, z_1\}$,
with horizontal slice $D_h = \{x_0, \dots, x_1\} \times \{z_0, \dots, z_1\}$,
heightmap $h : D_h \mapsto \mathbb{R}$,
ground layer $g : D_h \mapsto$ `GroundCell`,
surface layer $s : D_h \mapsto$ `SurfaceCell`,
atmosphere layer $f : D \mapsto$ `AtmosphereCell`.

`advect_temperature`(s, h)
`advect_water`(s, h)
`diffuse_quantities`(s)
for $\mathbf{p} \in D_h$ **do**
  $c \leftarrow \{f[\mathbf{p}.x, y, \mathbf{p}.z] : y \in \{y_0, \dots, y_1\}\}$
  `diffuse_atmosphere_to_surface`($s[\mathbf{p}]$, c)
  `diffuse_ground_to_surface`($s[\mathbf{p}]$, $g[\mathbf{p}]$)
  `apply_state_transitions`($s[\mathbf{p}]$, $h[\mathbf{p}]$)
  `diffuse_surface_to_atmosphere`($s[\mathbf{p}]$, c)

We model terrain-atmosphere interactions as a diffusion process consisting of three layers: a two-dimensional ground layer, a two-dimensional surface layer and a three-dimensional atmosphere layer. Table 1 describes the quantities stored in the cells of each layer.

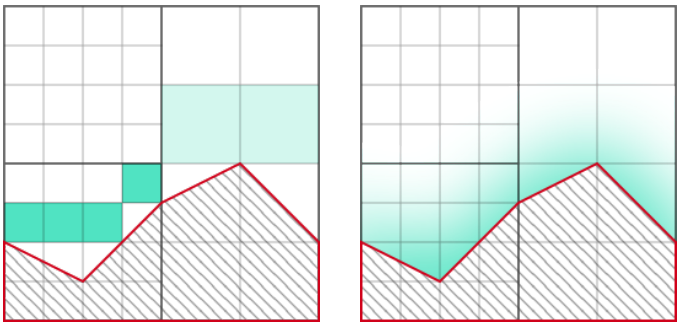
We update quantities in the three layers according to algorithm 6. First, we update the surface layer in isolation. The surface layer update transports temperature uphill, and water content downhill using semi-Lagrangian advection. It also diffuses both temperature and water content in each cell with their neighbors. The state transitions operation handles

microphysics per surface cell. These microphysics include the melting of snow, the pyrolysis of ground fuels, and other effects.

The ground layer models the base values. The quantities in the surface layer will revert to these base values when not externally influenced. We modeled interactions between the surface and the atmosphere layer as a diffusion process split into two parts. First, the surface layer is updated using quantities from the atmosphere. Then, the atmosphere is updated by diffusing quantities from the surface layer into the atmosphere. Since we work with fluid cells of different resolutions, diffusing the quantities into a single fluid cell would lead to unrealistic results, especially as cells get coarser (figure 13a). Instead, we diffuse the quantities from the surface into the complete column above the surface cell. We distribute the diffusion strength s following an exponential distribution (figure 13b). Let $\Delta a \geq 0$ be the altitude of a point above the surface. Let $\Delta t > 0$ be the timestep and let $h_d > 0$ be a general diffusion parameter, controlling the strength of the diffusion. Now $s(\Delta a) = \exp\left(-\Delta a/(h_d \Delta t)\right)$. Note that $\int_0^\infty s(\Delta a)\, d(\Delta a) = 1$ for all values of h_d and Δt.

Let c be an atmosphere cell with top and bottom altitudes b, respectively a. Let h be the altitude of the terrain under the cell. To calculate the diffusion strength for s_c for cell c, we integrate s over $[\max\{0, a-h\}, \max\{0, b-h\}]$. This results in the following formula:

$$s_c = \exp\big(-h_d \Delta t \max\{0, a-h\}\big) - \exp\big(-h_d \Delta t \max\{0, b-h\}\big). \tag{11}$$



(a) Diffusion into lowest cell that is completely above the terrain. Note the unnatural gaps.

(b) Ideal situation. Diffusion into all cells above the terrain following an exponential distribution.

Figure 13: Two different approaches to surface to atmosphere diffusion.

To diffuse values from the atmosphere into the surface, we apply the inverse operation of diffusion from the surface into the atmosphere. We sample quantities from the atmosphere column using the same distribution $s(\Delta a)$ and update the surface cells according to these quantities.

6.2 Expressiveness of our approach

With our approach, we produced a spatial transition from low fog-like structures to cumulus clouds (figure 14a). Contrary to the previous work (Hädrich et al. 2020), we show that this structure is unstable over time (figure 14b). The discrete patches of cloud visible at timestep $t = 150$ converge into a more uniform layer of clouds at timestep $t = 450$. This instability is caused by our multigrid projection method. Compared to the non-multigrid solver used in the previous work, our multigrid solver takes more global features of the velocity field into account.

Additionally, our approach captures rapidly evolving terrain conditions that the previous method could not capture, such as wildfires (figure 15).

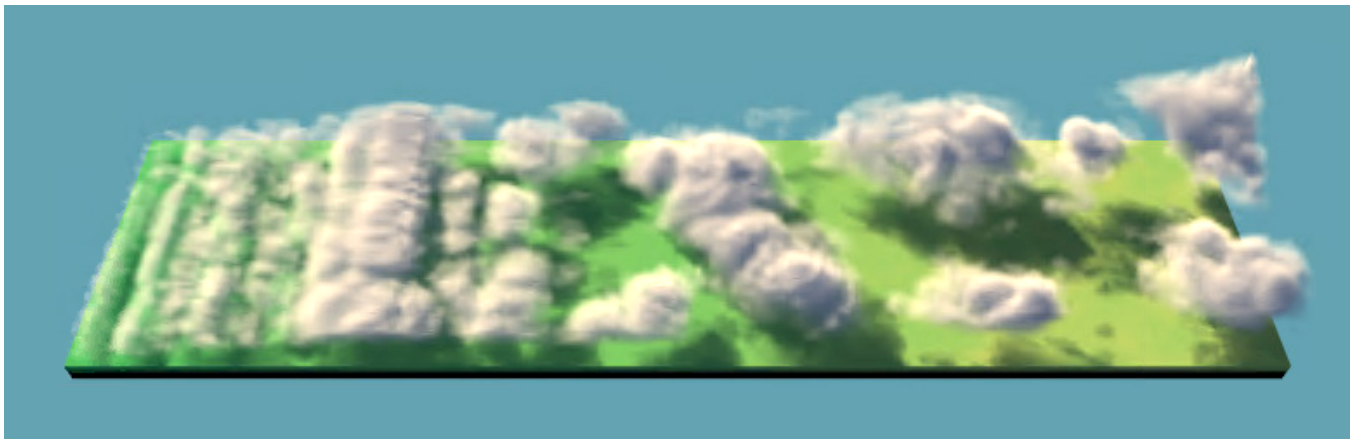
7 Results

In this section, we present the results of comparing our framework with other fluid systems. We implemented all simulations on DCGrid on the GPU using CUDA. Unless stated otherwise, we performed all simulations on an NVIDIA® GeForce GTX 1070.

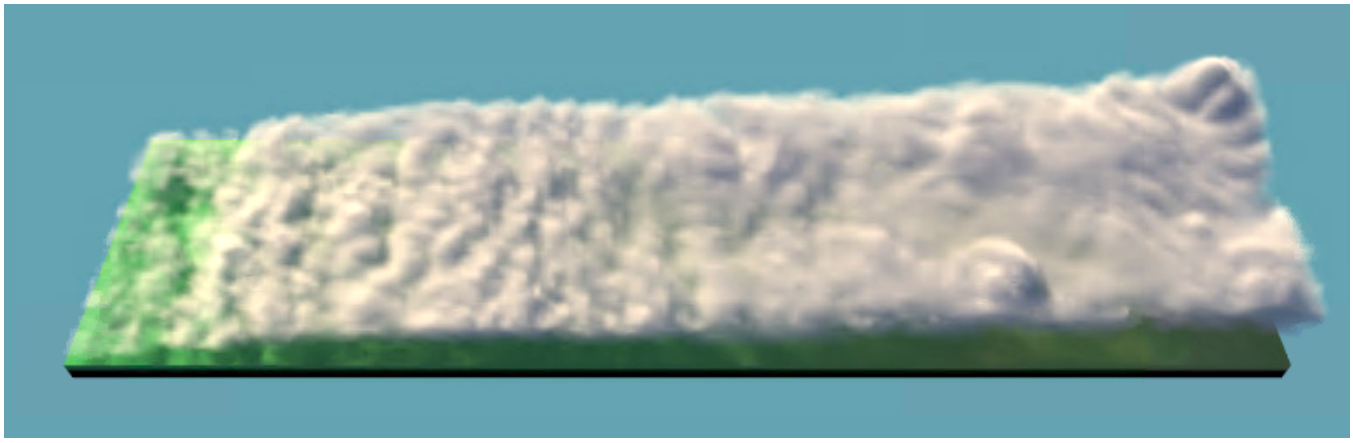
7.1 Comparison to a uniform grid

We evaluated the performance of the DCGrid data structure by running the same fluid simulation on a DCGrid instance in which the domain is densely allocated and on a simple uniform grid.

In terms of runtime, the DCGrid instance required 75% more time per frame than the uniform grid (table 2). Fluid operations on DCGrid are slower because they require pointer indirections. Also, after each update, DCGrid requires some extra time to accumulate changed values. Rendering performs worse because of the number of computations required to translate a position into a cell index.



(a) $t = 150$



(b) $t = 450$

Figure 14: Transition from fog (left) over stratocumulus to cumulus (right). We use the same setup as Hädrich et al. 2020.

DCGrid also has a memory overhead when compared to a uniform grid. In this example, DCGrid uses about 2.2 times the amount of memory that a uniform grid uses. In general, the memory overhead of DCGrid depends on the number of floating-point channels used per cell. When using 17 floating-point channels per cell, the memory overhead of DCGrid is only about 40%. The apron cell indices are the main contributor to the memory overhead.

7.2 Comparison to SPGrid

DCGrid is inspired by and shares characteristics with SPGrid (Setaluri et al. 2014). To test DCGrid against SPGrid, we recreated one of the scenes that they showcased (figure 16). Because of limited GPU memory, we can only allocate 112M cells in our DCGrid instance, compared to the 135M that SPGrid used. Even with fewer cells, our automatic topology adaptation scheme produces a more detailed simulation. Because DCGrid uses cell-centered advection, reuses temporary channels, and does not require ghost cells, DCGrid uses only about one-third of the memory compared to SPGrid. Even including the time for rendering, this experiment on DCGrid runs about 449 times faster than the original experiment on SPGrid (table 2). As DCGrid and SPGrid run on different systems and our experiment in DCGrid used only about 82% of the cells that the SPGrid experiment used, the performance difference might be less significant in other scenarios.

7.3 Comparison to GVDB

To compare DCGrid to GVDB, we set up a simple smoke simulation in GVDB (figure 17). In the experiment, we used cell-centered semi-Lagrangian advection, simple vorticity confinement, and non-multigrid projection using 10 Jacobi iterations for both DCGrid and GVDB. We set up GVDB such that the domain automatically expands around the smoke, once using bricks of 32^3 cells and once using bricks of 8^3 cells. GVDB used the least memory and ran the fastest using bricks of 8^3 cells. To get a fair comparison between GVDB and DCGrid, we configured the memory limit for the DCGrid instance to be equal to the memory used by the GVDB instance with bricks of 8^3 cells. Using this same amount of memory, DCGrid allocated 1.57M cells (of which 1.38M active cells), while GVDB allocated only 1.18M cells. Also, DCGrid performed the fluid simulation about 4.8 times as fast as GVDB overall. We attribute a large part of the performance difference between GVDB and DCGrid to the projection operation, where GVDB has to synchronize its apron cells after each iteration. For just rendering, GVDB

(a) $t = 150$, previous method (b) $t = 150$, our method (c) $t = 450$, our method

Figure 15: Wildfire caused by the ignition of dry grass.

Table 2: Timings and memory usage of experiments included in this work. For the quantities used in each experiment, see appendix A.

Scene	Structure	Domain	Memory (MB)	Floats / Cell	Avg. Time / Timestep (ms) Advect	Project	Topology	Render	Rest	Total
Fig. 7a	Uniform	256^3	537	8	34	24	-	2.0	16	76
Fig. 7b	DCGrid	256^3	1177	8	50	39	8.6	9.5	22	130
Fig. 7c	DCGrid	256^3	294	8	13	9.7	7.3	12	5.7	47
Fig. 7d	DCGrid	256^3	59	8	3.9	2.5	2.8	10	1.8	21
Fig. 7e	DCGrid	256^3	12	8	1.4	0.9	1.6	7.1	1.4	12
Fig. 1	DCGrid	512×128^2	193	17	11	4.8	6.4	25	13	60
Fig. 14	DCGrid	$512 \times 64 \times 128$	193	17	11	4.7	5.5	19	13	52
Fig. 15	DCGrid	256^3	193	17	9.7	4.4	5.1	17	9.5	46
Fig. 16a	SPGrid[1]	$1024^2 \times 2048$	23GB	16	26s	525s	-	?	24	575s
Fig. 16b	DCGrid	$1024^2 \times 2048$	6872	8	507	430	303	41	-	1280
Fig. 17a	GVDB	256^3	422	8	24	127	6.9	2.4	20	180
Fig. 17b	GVDB	256^3	97	8	15	102	9.3	1.4	15	142
Fig. 17c	DCGrid	256^3	97	8	5.2	9.1	3.5	9.2	2.2	29

[1] Data for the SPGrid result is sourced from Setaluri et al. 2014. This data did not include rendering time. Experiment run on Intel Xeon E5-2670.

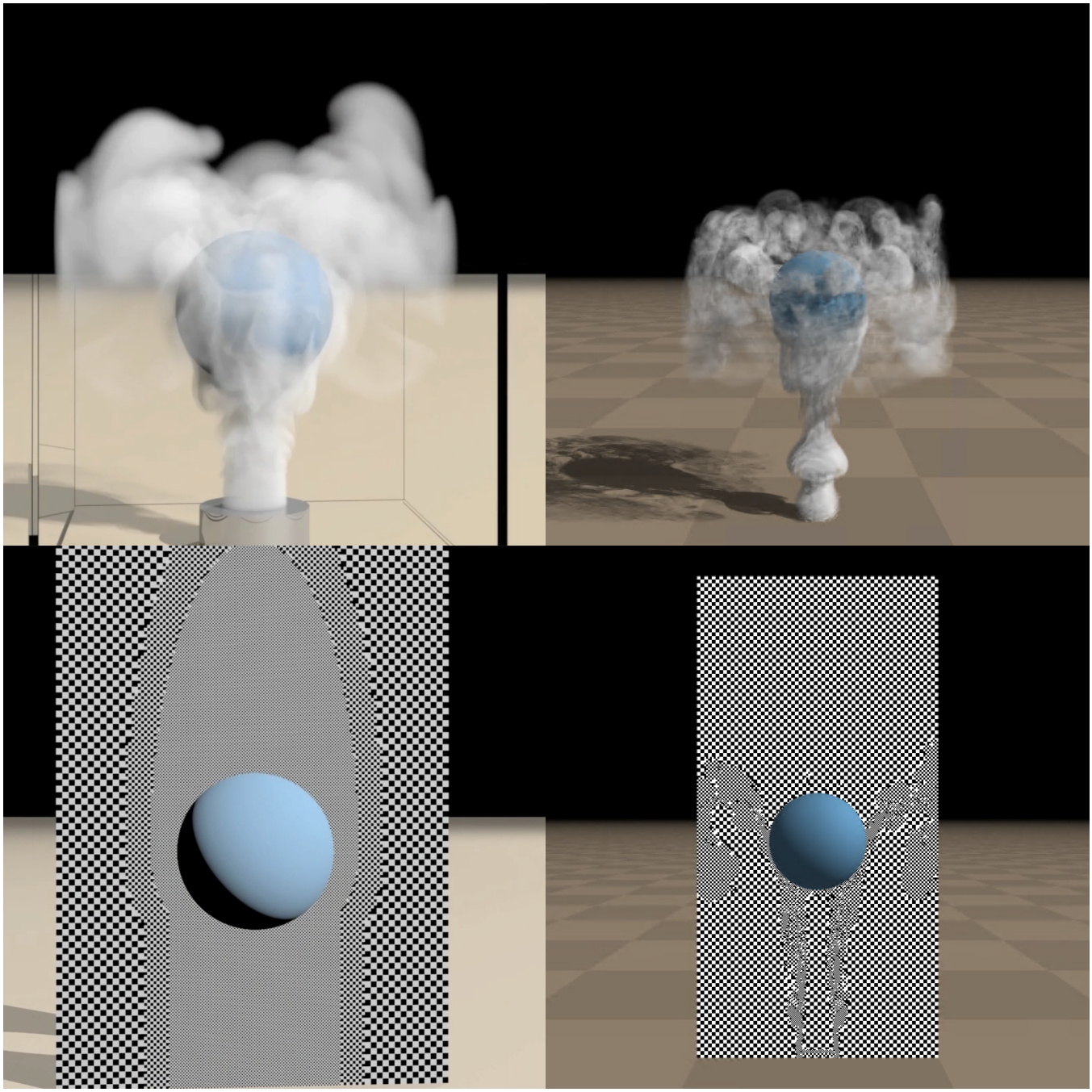
is significantly faster than DCGrid. We can attribute this performance to their efficient empty skipping and block traversal algorithms.

GVDB only allocates cells on the highest resolution. In this particular case, this benefits the result. GVDB only has to use memory for the high-resolution cells. Therefore, it can allocate more of them than DCGrid, using the same amount of memory. As a result, it preserves some more details in the simulation. In general, only allowing cells on the highest resolution can be a limitation for GVDB. Due to this restriction, for example, we have to resort to a non-multigrid solver for velocity projection.

8 Conclusions and Future Work

We presented a spatially adaptive grid structure tailored for fluid simulations and suitable for implementation on the GPU. We introduced an efficient, optimization-based algorithm for automatic local grid refinement based on user parameters. We also introduced new methods for efficiently handling solid-fluid interactions approximately for cells of different resolutions. Additionally, we were able to extend an earlier model for terrain-atmosphere interaction such that it handles rapidly changing values and a multi-resolution fluid grid. We demonstrated the expressiveness of this extension by recreating one scene with a transition from low-hanging, fog-like clouds to cumulus clouds and another with a wildfire. We compared our method to the state-of-the-art data structures SPGrid and GVDB and found that our method performs fluid simulations multiple times faster than both SPGrid and GVDB. We found that GVDB achieves superior rendering performance.

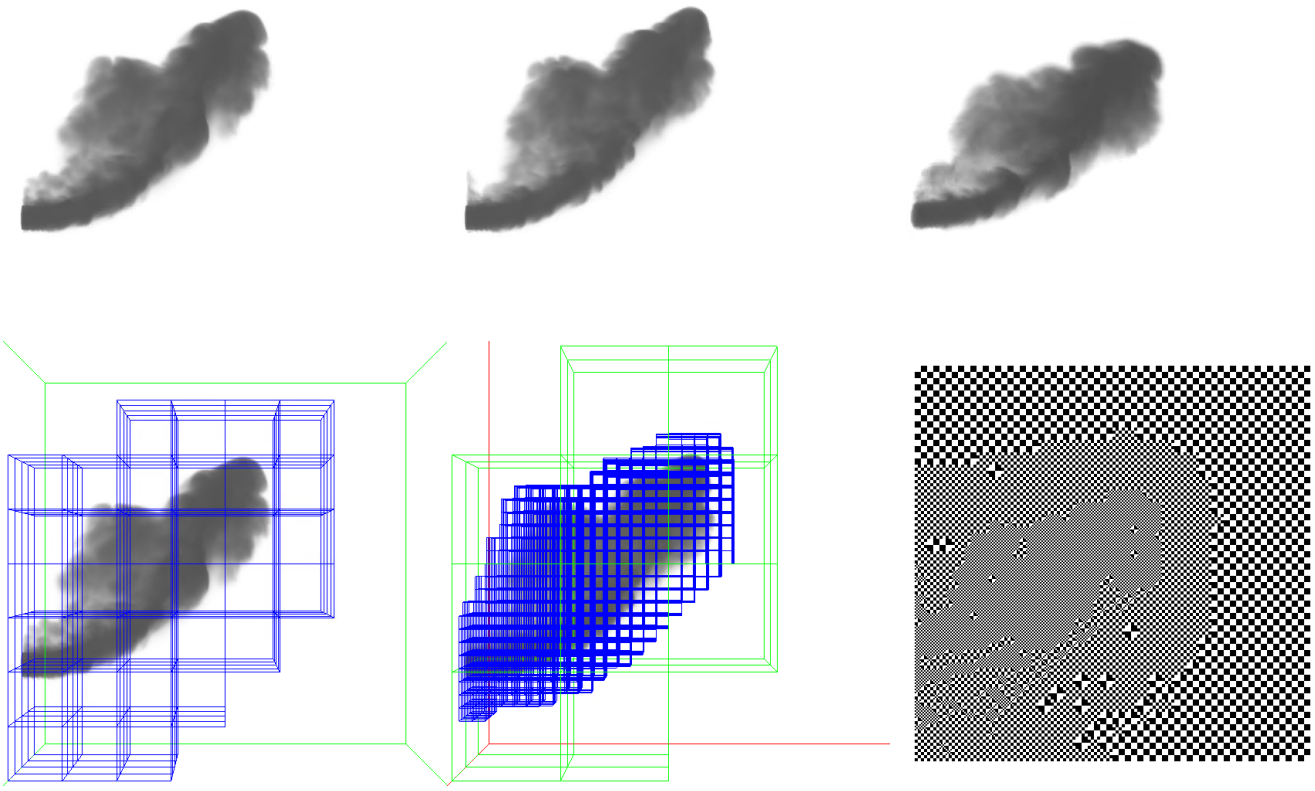
When comparing simulations on DCGrid to the same simulations on a uniform grid, we were able to generate similar results



(a) Smoke flow simulated using SPGrid, 135M allocated cells. Image source Setaluri et al. 2014.

(b) Smoke flow simulated using DCGrid, 112M allocated cells.

Figure 16: Comparison between DCGrid and SPGrid on a smoke flow around sphere. Domain size is $1024^2 \times 2048$, each checkerboard square represents a 4^3 cells.



(a) GVDB, brick size 32^3 (b) GVDB, brick size 8^3 (c) DCGrid

Figure 17: Smoke simulation run in GVDB and DCGrid on a 256^3 grid. Middle and right scene both use 97MB memory at most.

using only a fraction of the memory. Currently, densely allocated DCGrid instances suffer a performance and memory penalty compared to fluid simulations on a uniform grid. To reduce this performance penalty, it seems promising to explore CUDA's virtual memory management APIs, introduced in CUDA 10.2. If we can employ these APIs successfully, we could perform random access without relying on pointer indirections and hash table lookups, as if the grid were uniform. Consequently, the hash table and the pre-calculated apron cell indices would become redundant. Removing these two structures would significantly reduce both the runtime and the memory overhead. Additionally, we could explore the approach used in Wu et al. 2018 and store cell data in textures. Using textures could potentially improve the performance of advection and rendering, as it would enable hardware-accelerated interpolation.

Our structure is currently limited in three ways. First, we use fixed-size blocks of 4^3 cells. Allowing users to vary the block size could be beneficial to simulations. Larger blocks, for example, would require fewer apron cell indices per cell, which would reduce memory overhead. Second, we limit the maximum number of blocks per mipmap level. This limit does not appear to restrict the scenarios we explored. However, we could explore using an overall block limit instead. Such a limit would allow for more flexible grid layouts, which could help in highly dynamic scenarios. Finally, we only support rectangular domains, which limits the range of our simulations. As fluids often flow outside their initial domain, allocating cells outside the initially defined rectangle would open up new possibilities.

References

Aanjaneya, Mridul, Ming Gao, et al. (2017). "Power diagrams and sparse paged grids for high resolution adaptive liquids". In: *ACM Transactions on Graphics* 36.4, pp. 1–12. DOI: 10.1145/3072959.3073625.

Aanjaneya, Mridul, Chengguizi Han, et al. (2019). "An Efficient Geometric Multigrid Solver for Viscous Liquids". In: *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 2.2, pp. 1–21. DOI: 10.1145/3340255.

Blum, Manuel et al. (1973). "Time bounds for selection". In: *J. Comput. Syst. Sci.* 7.4, pp. 448–461.

Bouthors, Antoine and Fabrice Neyret (Aug. 2004). "Modeling clouds shape". In: *Eurographics (short papers)*. Eurographics Association. New Zealand: Eurographics Association. DOI: 10.2312/egs.20041020.

Brandt, Christopher et al. (2019). "The reduced immersed method for real-time fluid-elastic solid interaction and contact simulation". In: *ACM Transactions on Graphics* 38.6, pp. 1–16. DOI: 10.1145/3355089.3356496.

Cooke, J.J. et al. (2014). "Adaptive mesh refinement of gas–liquid flow on an inclined plane". In: *Computers & Chemical Engineering* 60, pp. 297–306. ISSN: 0098-1354. DOI: https://doi.org/10.1016/j.compchemeng.2013.09.007. URL: https://www.sciencedirect.com/science/article/pii/S0098135413002767.

Cornelis, Jens et al. (2014). "IISPH-FLIP for incompressible fluids". In: *Computer Graphics Forum* 33.2, pp. 255–262. DOI: 10.1111/cgf.12324.

Fedkiw, Ronald, Jos Stam, and Henrik Wann Jensen (2001). "Visual simulation of smoke". In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques - SIGGRAPH 01*. DOI: 10.1145/383259.383260.

Ferreira Barbosa, Charles Welton, Yoshinori Dobashi, and Tsuyoshi Yamamoto (2015). "Adaptive cloud simulation using position based fluids". In: *Computer Animation and Virtual Worlds* 26.3-4, pp. 367–375. DOI: 10.1002/cav.1657.

Ferstl, Florian et al. (2016). "Narrow Band FLIP for Liquid Simulations". In: *Computer Graphics Forum* 35.2, pp. 225–232. DOI: 10.1111/cgf.12825.

Foster, Nick and Ronald Fedkiw (2001). "Practical animation of liquids". In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pp. 23–30.

Gao, Ming et al. (2019). "GPU optimization of material point methods". In: *ACM Transactions on Graphics* 37.6, pp. 1–12. DOI: 10.1145/3272127.3275044.

Gardner, Geoffrey Y. (July 1985). "Visual Simulation of Clouds". In: *SIGGRAPH Comput. Graph.* 19.3, pp. 297–304. ISSN: 0097-8930. DOI: 10.1145/325165.325248. URL: https://doi.org/10.1145/325165.325248.

Gissler, Christoph et al. (2019). "Interlinked SPH Pressure Solvers for Strong Fluid-Rigid Coupling". In: *ACM Transactions on Graphics* 38.1, pp. 1–13. DOI: 10.1145/3284980.

Hädrich, Torsten et al. (2020). "Stormscapes: Simulating Cloud Dynamics in the Now". In: *ACM Transactions on Graphics* 39.6, pp. 1–16. DOI: 10.1145/3414685.3417801.

Hoetzlein, Rama Karl (2016). "GVDB: Raytracing Sparse Voxel Database Structures on the GPU". In: *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*. Ed. by Ulf Assarsson and Warren Hunt. ?: The Eurographics Association. ISBN: 978-3-03868-008-6. DOI: 10.2312/hpg.20161197.

Hu, Yuanming et al. (2018). "A moving least squares material point method with displacement discontinuity and two-way rigid body coupling". In: *ACM Transactions on Graphics* 37.4, pp. 1–14. DOI: 10.1145/3197517.3201293.

Inglis, T. et al. (2017). "Primal-Dual Optimization for Fluids". In: *Computer Graphics Forum* 36.8, pp. 354–368. DOI: 10.1111/cgf.13084.

Kajiya, James T. and Brian P Von Herzen (1984). "Ray Tracing Volume Densities". In: *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: Association for Computing Machinery, pp. 165–174. ISBN: 0897911385. DOI: 10.1145/800031.808594. URL: https://doi.org/10.1145/800031.808594.

Kim, Byungsoo et al. (2019). "Transport-based neural style transfer for smoke simulations". In: *ACM Transactions on Graphics* 38.6, pp. 1–11. DOI: 10.1145/3355089.3356560.

Lapointe, Caelan et al. (2020). "Efficient simulation of turbulent diffusion flames in OpenFOAM using adaptive mesh refinement". In: *Fire Safety Journal* 111, p. 102934. DOI: 10.1016/j.firesaf.2019.102934.

Lentine, Michael, Mridul Aanjaneya, and Ronald Fedkiw (2011). "Mass and Momentum Conservation for Fluid Simulation". In: *Proceedings of the 2011 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. SCA '11. Vancouver, British Columbia, Canada: Association for Computing Machinery, pp. 91–100. ISBN: 9781450309233. DOI: 10.1145/2019406.2019419. URL: https://doi.org/10.1145/2019406.2019419.

Liu, Haixiang et al. (Nov. 2016). "A Scalable Schur-Complement Fluids Solver for Heterogeneous Compute Platforms". In: *ACM Trans. Graph.* 35.6. ISSN: 0730-0301. DOI: 10.1145/2980179.2982430. URL: https://doi.org/10.1145/2980179.2982430.

Losasso, Frank, Frédéric Gibou, and Ron Fedkiw (2004). "Simulating water and smoke with an octree data structure". In: *ACM Transactions on Graphics* 23.3, pp. 457–462. DOI: 10.1145/1015706.1015745.

Museth, Ken (2013). "VDB: High-Resolution Sparse Volumes with Dynamic Topology". In: *ACM Transactions on Graphics* 32.3, pp. 1–22. DOI: 10.1145/2487228.2487235.

Neyret, Fabrice (1997). "Qualitative Simulation of Convective Cloud Formation and Evolution". In: *Eurographics Computer Animation and Simulation '97*. Vienna: Springer Vienna, pp. 113–124. ISBN: 978-3-7091-6874-5. DOI: 10.1007/978-3-7091-6874-5_8.

Nielsen, Michael B. and Robert Bridson (2016). "Spatially Adaptive FLIP Fluid Simulations in Bifrost". In: *ACM SIGGRAPH 2016 Talks*. SIGGRAPH '16. Anaheim, California: Association for Computing Machinery. ISBN: 9781450342827. DOI: 10.1145/2897839.2927399. URL: https://doi.org/10.1145/2897839.2927399.

Nielsen, Michael B., Konstantinos Stamatelos, et al. (2020). "Auto-Adaptivity: An Optimization-Based Approach to Spatial Adaptivity for Smoke Simulations". In: *Special Interest Group on Computer Graphics and Interactive Techniques Conference Talks* ? DOI: 10.1145/3388767.3407320.

Sato, Syuhei et al. (2018). "Example-based turbulence style transfer". In: *ACM Transactions on Graphics* 37.4, pp. 1–9. DOI: 10.1145/3197517.3201398.

Schoentgen, Arnaud et al. (2020). "A density-accurate tracking solution for smoke upresolution". In: *The Visual Computer* 36.10-12, pp. 2299–2311. DOI: 10.1007/s00371-020-01889-3.

Setaluri, Rajsekhar et al. (2014). "SPGrid: A Sparse Paged Grid structure applied to adaptive smoke simulation". In: *ACM Transactions on Graphics* 33.6, pp. 1–12. DOI: 10.1145/2661229.2661269.

Stam, Jos (1999). "Stable Fluids". In: *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '99. USA: ACM Press/Addison-Wesley Publishing Co., pp. 121–128. ISBN: 0201485605. DOI: 10.1145/311535.311548. URL: https://doi.org/10.1145/311535.311548.

Ummenhofer, Benjamin et al. (2019). "Lagrangian Fluid Simulation with Continuous Convolutions". In: *International Conference on Learning Representations*. ?: International Conference on Learning Representations.

Vimont, Ulysse et al. (2020). "Interactive Meso-scale Simulation of Skyscapes". In: *Computer Graphics Forum* 39.2, pp. 585–596. DOI: 10.1111/cgf.13954.

Wu, Kui et al. (2018). "Fast Fluid Simulations with Sparse Volumes on the GPU". In: *Computer Graphics Forum* 37.2, pp. 157–167. DOI: 10.1111/cgf.13350.

Xiao, Yuwei et al. (2020). "An adaptive staggered-tilted grid for incompressible flow simulation". In: *ACM Transactions on Graphics* 39.6, pp. 1–15. DOI: 10.1145/3414685.3417837.

Xie, You et al. (2018). "tempoGAN: A Temporally Coherent, Volumetric GAN for Super-resolution Fluid Flow". In: *ACM Transactions on Graphics* 37.4, pp. 1–15. DOI: 10.1145/3197517.3201304.

Zehnder, Jonas, Rahul Narain, and Bernhard Thomaszewski (2018). "An advection-reflection solver for detail-preserving fluid simulation". In: *ACM Transactions on Graphics* 37.4, pp. 1–8. DOI: 10.1145/3197517.3201324.

Zhang, Xinxin, Robert Bridson, and Chen Greif (2015). "Restoring the missing vorticity in advection-projection fluid solvers". In: *ACM Transactions on Graphics* 34.4, pp. 1–8. DOI: 10.1145/2766982.

Zhu, Yongning and Robert Bridson (2005). "Animating sand as a fluid". In: *ACM Transactions on Graphics (TOG)* 24.3, pp. 965–972.

A Quantities per Simulation

The quantities simulated in each experiment are summarized in table 3.

Table 3: Quantities simulated throughout the different experiments. Temporaries are reused for different purposes across multiple operations.

Scenes	Densities	Misc. quantities	Temporaries
Fig. 1, 14, 15	Vapor, Warm cloud, Frozen cloud, Ash, Rain, Snow, Graupel	Temperature, Fluidity	5
Fig. 16	Smoke	Fluidity	3
Fig. 7, 17	Smoke	Temperature	3

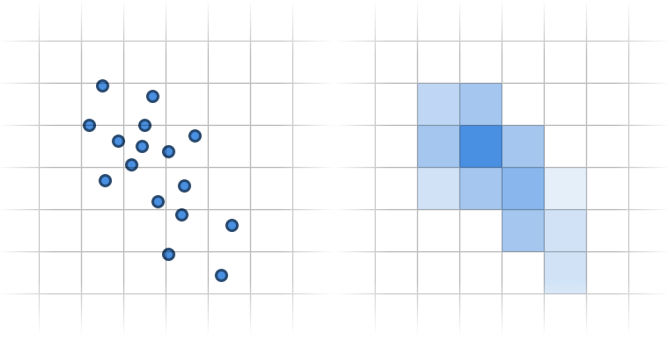
Supplementary Material

This section of the thesis aims to familiarize people with the topics discussed in the article. Reading this supplement should give you enough background information to fully appreciate the work presented in the article. This supplement also discusses alternative methods for spatial adaptivity in more depth.

1 Fluid Simulations

Fluid simulation is a broad term, that encapsulates the simulation of any sort of liquid or gas, their interactions with each other, and their interactions with other objects. On a basic level, fluid flows can be described from two perspectives. On one side, fluids can be seen as a large collection of particles moving through space. Each particle has its individual mass, velocity, temperature and other properties. This perspective is called the Lagrangian perspective (figure 18a). On the other side, we can divide a space into cells. Instead of describing how each individual particle moves, instead, we describe how the fluid at each cell evolves over time. This perspective is called the Eulerian perspective (figure 18b). Fluid simulation methods often mix these two perspectives to reach good results.

In this thesis, we focus on simulating fluids as seen from an Eulerian perspective. This section aims to give an overview of the basic aspects of such a simulation. For clarity, we focus on incompressible, homogeneous, single-phase fluids, on a two-dimensional, Cartesian grid with spacing 1 and leave out diffusion, vorticity confinement and external forces. For a more complete introduction to fluid simulation on Cartesian grids, see Stam 1999.



(a) Lagrangian perspective (b) Eulerian perspective

Figure 18: The same fluid described from two different perspectives.

1.1 Navier-Stokes Equations for Incompressible Flow

The state of a fluid at time t at position $\mathbf{x}$ in a Cartesian grid with spacing 1 can be described by a velocity field $\mathbf{u}(\mathbf{x}, t)$ and a pressure field $p(\mathbf{x}, t)$. If we know the initial velocity and pressure, then we can describe the evolution of these fields using the Navier-Stokes equations:

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \nabla p, \tag{12}$$

$$\nabla \cdot \mathbf{u} = 0. \tag{13}$$

To solve these equations, we use a version of the Helmholtz-Hodge decomposition theorem:

Theorem 2. *Let $D \subseteq \mathbb{R}^2$ be a bounded set with smooth boundary ∂D and let $n : \partial D \mapsto \mathbb{R}^2$ be the normal direction of this boundary. Let $\mathbf{w}$ be a vector field on D that is twice differentiable. Now $\mathbf{w}$ can be uniquely decomposed in a divergence free vector field $\mathbf{u}$ and in the gradient of a scalar field ∇p as*

$$\mathbf{w} = \mathbf{u} + \nabla p. \tag{14}$$

Next, we define the distributive projection operator P, that maps a vector field to its divergence-free part: $P\mathbf{w} = \mathbf{u}$.

If we apply P to equation 14, we find $P\mathbf{w} = P\mathbf{u} + P(\nabla p)$. Since $\mathbf{u}$ is already divergence free, $P\mathbf{u} = \mathbf{u}$ and thus $\mathbf{u} = \mathbf{u} + P(\nabla p)$. Therefor, $P(\nabla p) = 0$.

By applying the projection operator to both sides of equation 12, we get:

$$P\frac{\partial \mathbf{u}}{\partial t} = P(-(\mathbf{u} \cdot \nabla)\mathbf{u} - \nabla p) \tag{15}$$

As $\mathbf{u}$ is divergence free, so is $\frac{\partial \mathbf{u}}{\partial t}$, hence equation 15 simplifies to:

$$\frac{\partial \mathbf{u}}{\partial t} = P(-(\mathbf{u} \cdot \nabla)\mathbf{u}) = P\mathbf{w} \tag{16}$$

This formulation is very useful, as $\mathbf{w}$ is simply the resulting velocity field after advection of the velocity field by itself (section 1.2).

To determine $P\mathbf{w}$, we can apply the Helmholtz-Hodge decomposition 2:

$$\mathbf{w} = P\mathbf{w} + \nabla p. \tag{17}$$

If we apply the divergence operator on both sides of equation 17, we get:

$$\nabla \cdot \mathbf{w} = \nabla(P\mathbf{w} + \nabla p) = 0 + \nabla^2 p. \tag{18}$$

The last equation holds because $P\mathbf{w}$ is divergence-free. The resulting equation is a Poisson equation, also known as the pressure Poisson equation:

$$\nabla^2 p = \nabla \mathbf{w} \tag{19}$$

We can retrieve p by solving this equation, either analytically, or using numerical methods (1.3). Using the value for p, we can write $P\mathbf{w} = \mathbf{w} - \nabla p$ and we find:

$$\frac{\partial \mathbf{u}}{\partial t} = \mathbf{w} - \nabla p \tag{20}$$

1.2 Semi-Lagrangian Advection

Advection is the process with which quantities are propagated through space. Advection can be applied to densities, velocity and any other quantity of the fluid field.

To advect quantities, we use the first-order implicit integration method as proposed by Stam 1999. To advect a quantity q through the domain:

$$q_{t+\Delta t}(\mathbf{x}) = q_t(\mathbf{x} - \mathbf{u}(\mathbf{x}, t)\Delta t) \tag{21}$$

The advantage of this method over other methods, is that it is simple and stable for arbitrary timesteps and velocities.

To implement this method, we assume a particle to be present at the center of each grid cell $\mathbf{x}$ at time $t + \Delta t$ and we trace the trajectory of this particle back to where it was at time t, given $\mathbf{u}(\mathbf{x}, t)$. We then find the resulting quantity by linearly interpolating the quantities at time t at the 4 grid cells surrounding the backtraced position.

1.3 Solving the Pressure Poisson Equation

The Poisson equation $\nabla^2 p = \nabla \cdot \mathbf{w}$ can be written as a matrix equation $A\mathbf{x} = \mathbf{b}$ where $A = \nabla^2$, $\mathbf{x} = p$ and $\mathbf{b} = \nabla \cdot \mathbf{w}$. To solve this matrix equation, we need to find the inverse of A. The size of this matrix depends directly upon the resolution of the grid, and thus it quickly becomes infeasible to compute its inverse analytically as the grid resolution grows.

Instead, we can approach the solution using an iterative method. Our method of choice is the Jacobi iteration method, as it is easily paralellizable on the GPU.

The Jacobi method works by decomposing matrix A into a diagonal matrix D and a matrix R with diagonal elements equal to zero, such that $A = D + R$.

The solution to $A\mathbf{x} = \mathbf{b}$ is then approximated by:

$$\mathbf{x}^{(n+1)} = D^{-1}(\mathbf{b} - R\mathbf{x}^{(n)}) \tag{22}$$

For our pressure Poisson equation, computing one Jacobi iteration per grid cell now amounts to:

$$p_{i,j}^{(n+1)} = \frac{p_{i+1,j}^{(n)} + p_{i-1,j}^{(n)} + p_{i,j+1}^{(n)} + p_{i,j-1}^{(n)} - (\nabla \cdot \mathbf{w})_{i,j}}{4} \tag{23}$$

We can now solve the pressure Poisson equation by taking an initial guess of 0 everywhere. With a zero initial guess however, a lot of Jacobi iterations are required for the solution to converge.

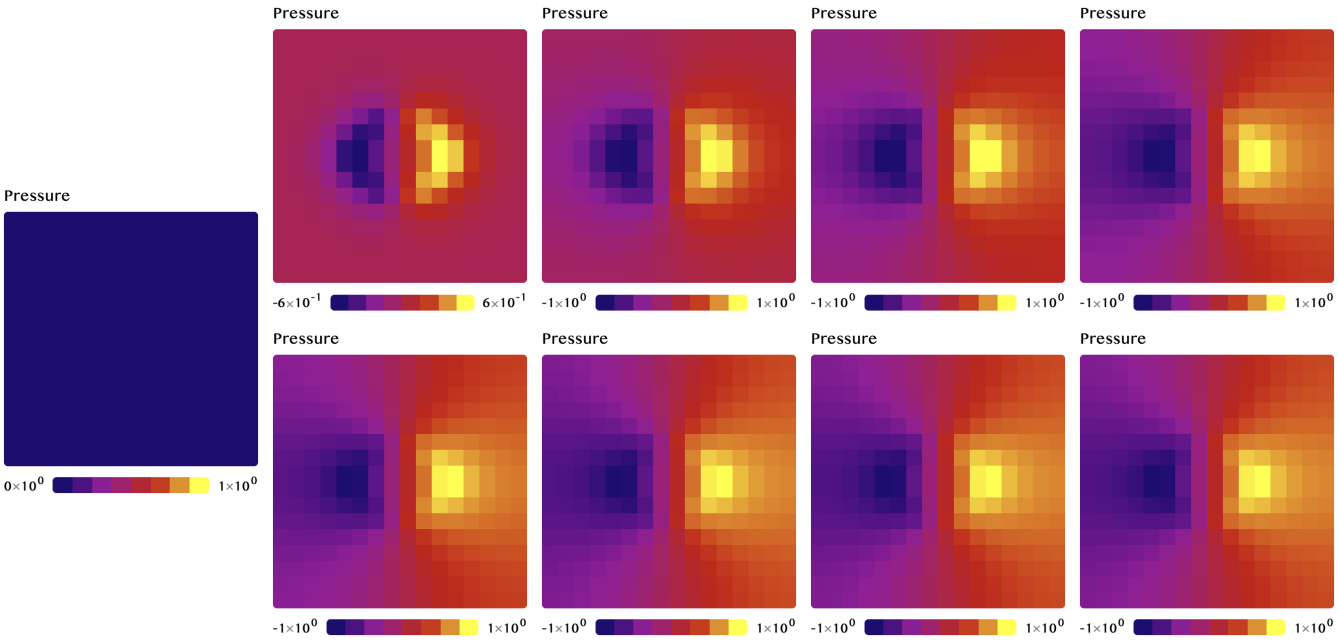
To decrease the number of iterations required for the Jacobi method to converge, we can use a different initial guess. One way of coming up with such an educated initial guess, is to use a multigrid solver.

A multigrid solver generally includes two phases: a down-stroke, and an up-stroke. In the down-stroke, the target problem is repeatedly restricted, or scaled down, from a $n \times n$ grid to a $n/2 \times n/2$ grid, until the grid size is as small as desired. At that point, the problem is solved on the small grid. In the upstroke, the solution on the small $n \times n$ grid is used as an initial guess for the solution to the problem on the $2n \times 2n$ grid. Then, the problem is solved on the $2n \times 2n$ using Jacobi iterations. This propagation routine is repeated until we arrive at the original grid again.

This way of solving matrix equation $A\mathbf{x} = \mathbf{b}$ requires way fewer iterations than taking 0 as the initial guess.

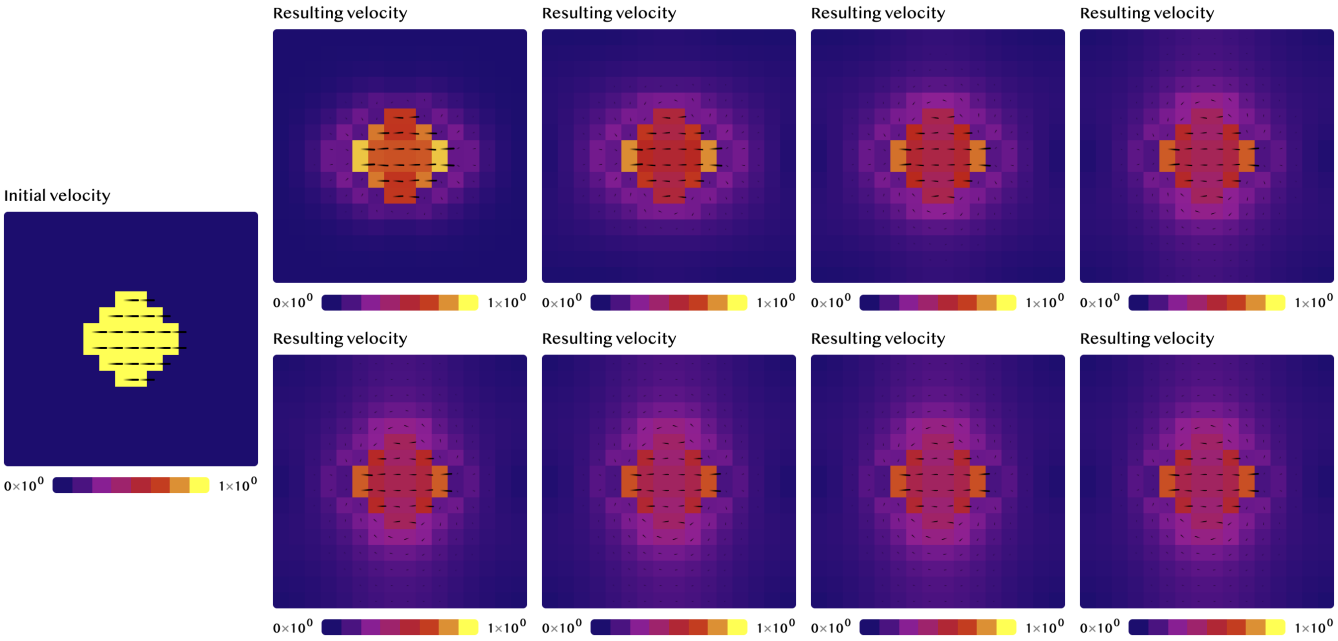
To illustrate the improved convergence rates, we set up an experiment. Figures 19, 20 and 21 show the behavior of a plain Jacobi solver versus a Jacobi solver using a multigrid method. After 200 iterations, still 7% of the original error is left in the solution reached by the plain Jacobi solver. In contrast, the multigrid pre-conditioned Jacobi solver reaches this same error level after only 12 iterations[1].

[1] For an interactive solver of the pressure Poisson equation, visit `https://wouterraateland.github.io/multigrid`.



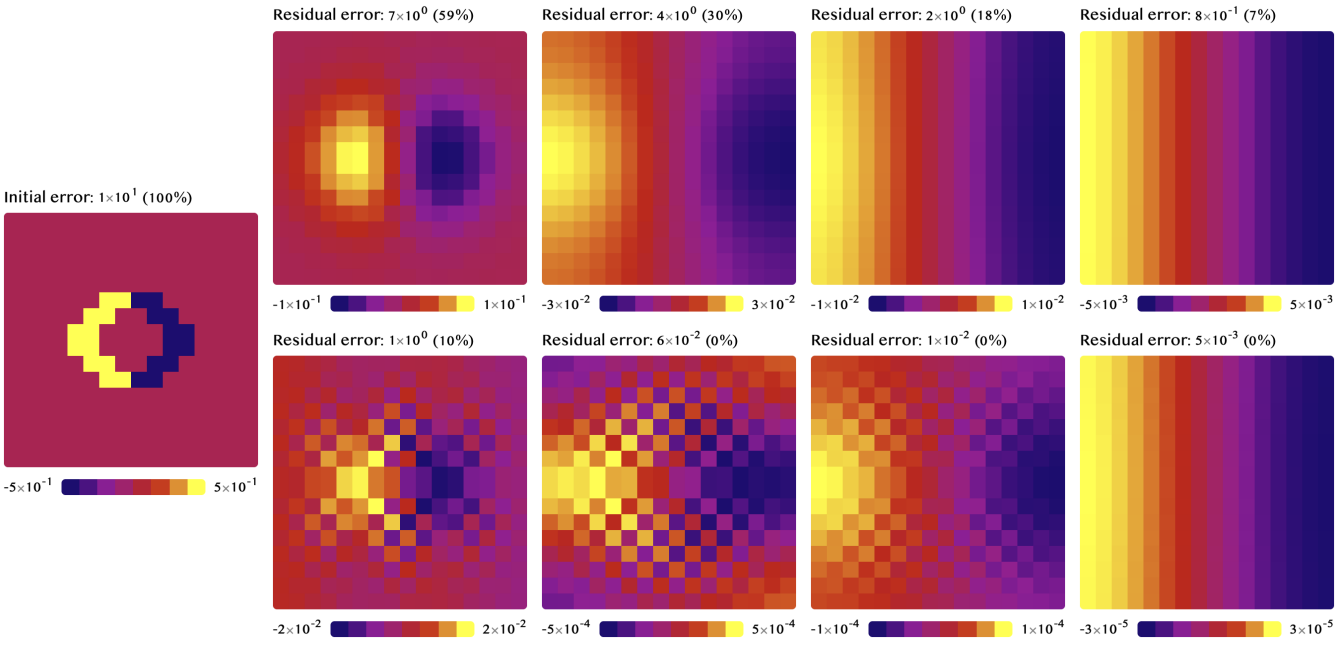
(a) 10 iterations (b) 50 iterations (c) 100 iterations (d) 200 iterations

Figure 19: Pressure field after different numbers of Jacobi iterations. Top row does not use multigrid, bottom row uses multigrid.



(a) 10 iterations (b) 50 iterations (c) 100 iterations (d) 200 iterations

Figure 20: Resulting velocity field after different numbers of Jacobi iterations. Top does not use multigrid, bottom row uses multigrid.



(a) 10 iterations (b) 50 iterations (c) 100 iterations (d) 200 iterations

Figure 21: Error after different numbers of Jacobi iterations. Top row does not use multigrid, bottom row uses multigrid.

2 Alternative Adaptive Grid Structures

In this section, we describe approaches taken by previous works to realize spatial adaptivity. In particular, we describe the approaches taken by GVDB, AST and OpenFOAM.

2.1 GVDB

GVDB is a GPU implementation of the OpenVDB data structure, which models a spatial domain using a tree with a high branching factor. The root node of the tree covers the complete domain, and the leaf nodes represent high-resolution voxels. GVDB achieves spatial adaptivity by allocating the tree partially. This partial allocation results in a domain sparsely populated with high-resolution voxels. GVDB includes two methods to activate regions of the spatial domain. Internally, both methods add nodes to the tree structure. In particular, these methods add a leaf node for each position in the region to activate. The first way to activate a region is to do so directly. The other way to activate a region is to pass GVDB a list of points on which the domain should be active. This way is helpful for hybrid particle/grid simulations such as FLIP, as it allows incremental updates in which the topology follows the fluid particles.

2.2 Adaptive Staggered-Tilted Grid

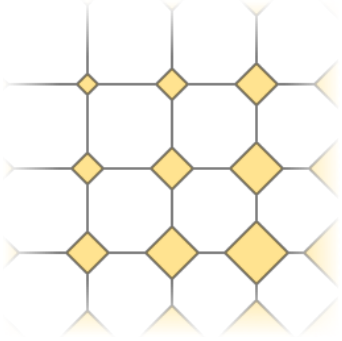


Figure 22: Adaptive Staggered-Tilted grid with varying adaptation states. Tilted cells marked in yellow.

Xiao et al. 2020 introduced an alternative approach that adds adaptive topology to uniform grids. It does so by augmenting the primary grid with a secondary, overlapping grid consisting of tilted cells (i.e., rotated by 45° in 2D). These tilted cells

can be closed (22, top left), opened (22, bottom right), or they can have any state in between. If all tilted cells surrounding a primary grid cell are open, they effectively half the size of this primary cell, thus achieving spatial adaptivity. This method can be combined with multigrid methods such as SPGrid and typically imposes a runtime penalty of only a few percent.

2.3 OpenFOAM

OpenFOAM is a simulation application used extensively in the computation of fluid flows for engineering purposes. It operates on meshes with cuboid cells instead of regular grids. OpenFOAM includes an adaptive mesh refinement (AMR) module that achieves an adaptive topology by refining and unrefining cells as follows: Before running a simulation, a user defines thresholds $t_0, \ldots, t_L$, a refinement property p and a maximum number of cells N. After each timestep, the AMR module iterates over each cuboid cell c at refinement level i. If $p(c) > t_i$, then it adds c to the list of cells to refine. Next, it picks the first n cells from that list, such that the total number of cells after refinement does not exceed N. Finally, it splits the selected cells into 8 "refined" cells. The unrefinement step selects all cells for which the value is below the user-defined thresholds. This method of topology adaptation has some caveats:

1. The user must know good values for the thresholds before running the simulation.

2. Instead of selecting the most important cells from the list with refinement candidates, the AMR module truncates it. Consequently, cells selected for refinement are not necessarily the best choice, leading to a potentially sub-optimal topology.

3 Signed Distance Functions

In computer graphics, we frequently represent geometry as collections of triangles. These collections of triangles can grow to billions, and graphics pipelines are optimized to handle them very rapidly.

In this work, we need to determine the minimum distance to the geometry for arbitrary points. If we were to represent the geometry as a collection of triangles, we would have to iterate over each of those triangles to determine this minimum distance. Finding the distance to the geometry this way would be a time-consuming process. So, instead of representing geometry as collections of triangles, we use signed distance fields (SDFs).

For any metric space (M, d), the SDF for closed set $\Omega \subseteq M$ is a function $f : M \mapsto \mathbb{R}$ such that

$$f_\Omega(\mathbf{x}) = \begin{cases} d(\mathbf{x}, \partial\Omega) & \text{if} \quad \mathbf{x} \in \Omega^c \\ -d(\mathbf{x}, \partial\Omega) & \text{if} \quad \mathbf{x} \in \Omega \end{cases} \tag{24}$$

Where $\partial\Omega$ denotes the boundary of Ω and

$$d(\mathbf{x}, \partial\Omega) = \inf_{\mathbf{y} \in \partial\Omega} d(\mathbf{x}, \mathbf{y}). \tag{25}$$

To illustrate this, consider $M = \mathbb{R}^2$, d the euclidean function and let S be the square with sides of length 6 centered around the origin (figure 23), then the SDF f_S for this square equals:

$$f_S(\mathbf{p}) = \begin{cases} \sqrt{\max\{0, |\mathbf{p}.x| - 3\}^2 + \max\{0, |\mathbf{p}.y| - 3\}^2} \\ \quad \text{if } \max\{\mathbf{p}.x, \mathbf{p}.y\} > 3 \\ \max\{|\mathbf{p}.x|, |\mathbf{p}.y|\} - 3 \\ \quad \text{otherwise} \end{cases} \tag{26}$$

Finding the distance from an arbitrary point to the geometry using SDFs requires way fewer operations than comparing with each triangle. Many geometric primitives have a simple SDF formulation. The properties of SDFs make it computationally cheap to perform transformations on them (i.e., scaling, repetition or merging) [2].

In this work, we use SDFs to render fluids, solids, and their shadows in a single ray marching framework (figure 24). Using the properties of SDFs, we were able to render soft shadows without any computational cost (figure 24c).

4 GPU Programming with CUDA

People frequently use GPUs to render 3D scenes. However, modern GPUs are capable of running any computation that CPUs can. In this section, we discuss the trade-offs between running computations on a GPU and CPU. We also explain basic concepts of GPU programming using CUDA.

Modern GPUs consist of many cores, which are each capable of running many operations in parallel. This architecture makes GPUs a perfect fit for operations that require many similar computations. There are some drawbacks to running operations on a GPU. One of the drawbacks is that GPUs and CPUs are different chips. Therefore, data needs to be transferred between them, which causes some runtime overhead. Also, in general, GPU cores have lower clock speeds than CPU cores and are optimized for floating-point operations. Therefore, operations that rely on integer or boolean arithmetic might be faster on the CPU.

[2] For an incredible application of SDFs, see `https://www.shadertoy.com/view/WsSBzh`.

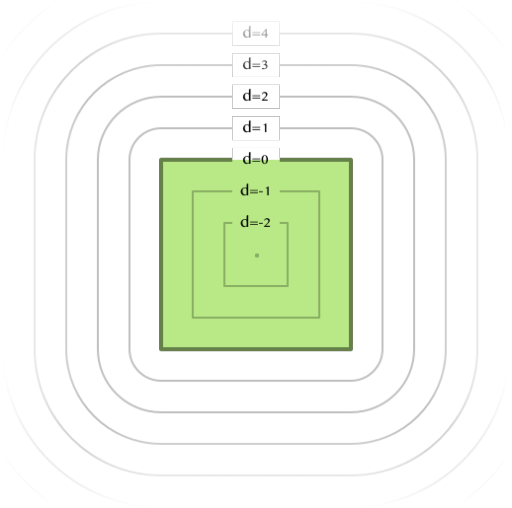
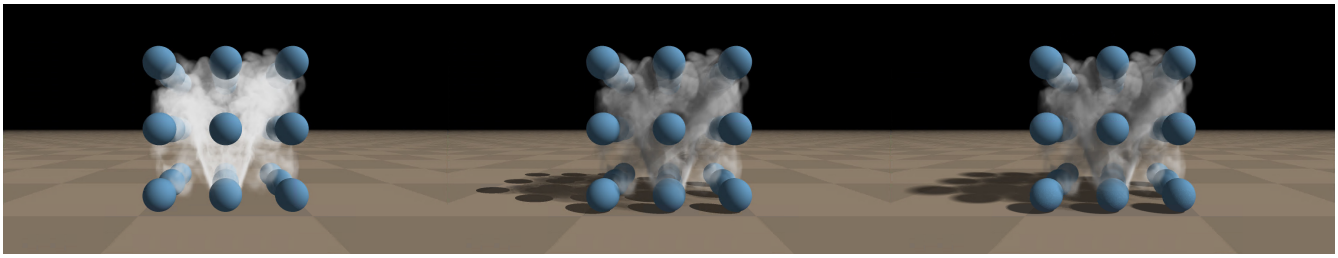


Figure 23: Visualization of the signed distance function for a square with sides of length 6 in $\mathbb{R}^2$.



(a) No shadows (b) Harsh shadows (c) Soft shadows

Figure 24: Using SDFs to render spheres and shadows.

CUDA is a platform that makes it possible to write programs running on the GPU using just C++ or some other language. To illustrate how this works, take the following C++ function that increments each element of an array:

```
void increment_array(float *a, int n) {
    for (int i = 0; i < n; i++)
        a[i]++;
}
```

Normally, we would run this function on the CPU using the following code:

```
int n = 1 << 15;
float *x = (float *)malloc(n * sizeof(float));
increment_array(x, n);
free(x);
```

To perform this operation on the GPU, we can write a CUDA kernel instead:

```
__global__ void k_increment_array(float *a, int n) {
    for (int i = 0; i < n; i++)
        a[i]++;
}
```

So far, the only change is the `__global__` prefix. This prefix indicated that a function should be called from the CPU and run on the GPU.

We would run this kernel using the following code:

```
float *x;
cudaMalloc(&x, n * sizeof(float));
k_increment_array<<<1,1>>>(x, n);
cudaFree(x);
```

Here, `cudaMalloc` and `cudaFree` are the GPU counterparts of `malloc` and `free`, and `<<<1,1>>>` indicates that our CUDA kernel should be run in 1 block, consisting of 1 thread. Hence, we are running on the GPU, but we don't benefit from any parallelization yet. To run our function in parallel, we can to modify our kernel such that each thread runs only a single increment, as follows:

```
__global__ void k_increment_array(float *a, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n)
        a[i]++;
}
```

We can now run this kernel using a parallel configuration:

```
int blockSize = 256;
int gridSize = (n + blockSize - 1) / blockSize;
k_increment_array<<<gridSize, blockSize>>>(x, n);
```

This configuration runs the kernel using blocks of 256 threads. We defined `gridSize` such that the GPU creates at least as many threads as there are array elements. A correctly configured parallel computation can perform more than 100 times faster than a serialized CPU computation.

Internally, thread blocks are composed of sets of 32 threads, called warps. Threads in a warp execute the same operations simultaneously. In some cases, we want different threads in the same warp to write to one memory location simultaneously.

Consider the following kernel, in which each thread increments a number:

```
__global__ void k_increment(int *value) {
    int previous = *value++;
    printf("Previous: %d\n", previous);
}
```

Running `k_increment` with one block of 32 threads would print 0, 32 times.

To increment the value 32 times, we have to use CUDA atomic operations. Atomic operations work in three steps. First, they restrict access to a memory location to a single thread. Then, they execute the actual operation. Finally, they release the memory again. This way, atomic operations enable serial computation on the GPU.

Using atomic operations, we can rewrite our increment kernel as follows:

```
__global__ void k_increment_atomic(int *value) {
    int previous = atomicAdd(value, 1);
    printf("Previous: %d\n", previous);
}
```

Running `k_increment_atomic` with one block of 32 threads would print values 0 up to 31, as expected.