OPTIMISATION OF A FLIP ALGORITHM

Christoph Amevor Sean Bone Felix Illes Mikael Stellio

Department of Computer Science
ETH Zurich, Switzerland

The following is the report for the project of group 32 in the FS21 Advanced Systems Lab lecture.

ABSTRACT

The Fluid Implicit Particle (FLIP) algorithm is commonly used in computer graphics to simulate fluids such as smoke or water. To improve the scale and realism of these simulations, we may wish to increase the number of cells and particles, which comes at the cost of higher memory usage and increased runtime. It is therefore desirable to optimise the implementation, in order to maximise the quality of the results that can be achieved in a given time. In this work, we focus on optimising an existing implementation of a FLIP algorithm; in particular, we design improved data structures and implement a highly optimised sparse linear systems solver which outperforms the previous *Eigen*-based implementation.

1. INTRODUCTION

A plethora of algorithms exist in the realm of computational fluid dynamics (CFD), each of which has its merits and drawbacks. Engineering applications tend to favor grid-based (Eulerian) methods, as these can achieve very high physical accuracy. In Computer Graphics however, we may wish to forego some accuracy in favour of greater artistic freedom and turnaround time. As a result, particle-based (Lagrangian) and hybrid (Lagriangian-Eulerian) methods such as FLIP are popular, since they can easily model fluid surfaces, are generally quite fast, and can still look appealing.

Motivation. A baseline *C++* implementation of a FLIP method was provided as part of a previous group project, involving two of the current team members, from the *"Physically-Based Simulation for Computer Graphics"* lecture at ETHZ. Simulating a 22-second video for that project took over 18 hours: optimising the code could allow for higher-quality results in the same time. Furthermore, FLIP is composed of multiple components, each of which could be optimised independently, making for an ideal group project.

Contribution. In this project, we aimed at developing a highly optimised version of the FLIP algorithm. We started from a baseline implementation that made extensive use of the *Eigen* library. We optimised with a focus on the Intel Skylake architecture and the Advanced Vector Extensions 2 (AVX2) instruction set. The optimised version now runs upwards of 8 times faster than the original implementation and requires considerably less memory. It is worth mentioning that, alongside other optimisations, we implemented a custom incomplete Cholesky conjugate gradient (ICCG) solver to better exploit the known structure of our specific problem. The new solver outperforms comparable options found in the *Eigen* library which were employed in the baseline implementation of the algorithm.

2. BACKGROUND

We begin by looking at the incompressible Navier-Stokes equations, which model the fluid flows we are interested in.

$$\frac{\partial \vec{u}}{\partial t} = \vec{F} + \nu \nabla^2 \vec{u} - \vec{u} \cdot \nabla \vec{u} - \frac{\nabla p}{\rho} \tag{1}$$

$$\nabla \cdot \vec{u} = 0 \tag{2}$$

Here $\vec{u}$ is the velocity field and $\vec{F}$ refers to the external forces such as gravity. The term $\nu \nabla^2 \vec{u}$ represents the diffusion of the fluid velocity field, and models the viscosity of the fluid. This term is commonly discarded in Computer Graphics applications, since the artificial diffusion introduced by discretisation errors usually more than makes up for its absence. The term $\vec{u} \cdot \nabla \vec{u}$ models self-advection, which represents the transport of velocity by the velocity field itself. The last term $\frac{\nabla p}{\rho}$ subtracts the pressure gradient adjusted to the density, which is generally taken to be constant. Finally, equation (2) imposes an important property of incompressible fluid flows, namely that the velocity field is divergence-free.

In general, there are two commonly used frameworks when modeling fluids: Lagrangian and Eulerian. A Lagrangian method will model the fluid with particles which conserve velocity, whereas an Eulerian approach will use a grid instead, to more accurately represent the velocity field of the fluid. The FLIP algorithm attempts to combine the strengths of both methods, and is termed a hybrid method because it combines the use of particles, used to keep track

of the fluid's surface, with the use of a marker-and-cell method (MAC) grid, used to impose the divergence-free property.

Operator Splitting. The FLIP algorithm solves the incompressible Navier-Stokes equations by operator splitting. In other words, equation (1) is separated into two steps which are applied in succession:

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} = \vec{F}, \tag{3}$$

$$\frac{\partial \vec{u}}{\partial t} = -\frac{\nabla p}{\rho}, \tag{4}$$

where the pressure field in (4) is calculated as follows to impose condition (2), resulting in the following Poisson equation:

$$\nabla^2 p = \nabla \cdot \vec{u}. \tag{5}$$

One timestep of the FLIP algorithm can be summarised as follows:

1. Compute the velocity field by projecting the velocities from the particles to the grid (particle-to-grid projection).
2. Apply external forces to the velocity field.
3. Enforce boundary conditions.
4. Compute pressure gradients by solving (5) and correct the velocity field to be divergence-free with (4).
5. Update particle velocities (grid-to-particle projection).
6. Advect particles using 2nd-order Runge-Kutta to solve equation (3).
7. Reconstruct the surface mesh of the fluid using a level set and the marching cubes algorithm.

2.1. Pressure Correction

To conserve mass we aim for a divergence-free velocity field. By solving the Poisson equation (5) we can calculate the pressure field, which we use to enforce a divergence-free velocity. The discretisation of the Poisson equation results in a large sparse linear system of equations (LSE) of the form $Ap = d$, where $A \in \mathbb{R}^{N_{\text{cells}} \times N_{\text{cells}}}$. We note that the original implementation defers solving the LSE to a library. The matrix A has a very simple structure: the diagonal depends solely on the domain geometry, thus it can be precomputed. Furthermore, all off-diagonal elements turn out to be either 0 or -1, and could be easily computed on-the-fly.

PCG algorithm. Preconditioned Conjugate Gradient (PCG) algorithms lend themselves to solving such systems. Although conjugate gradient algorithms may take up to N steps in general, convergence is much quicker given a suitable preconditioning matrix M and solving the preconditioned LSE $MAp = Md$ instead. Pseudocode for the PCG algorithm is shown in Algorithm 1, where the function `applyM`(r) consists of the matrix-vector product Mr in symbolic terms (and `applyA`(s) represents As). For efficiency and stability reasons, instead of computing the inverted matrix M, practical implementations use forward and backward substitutions to solve the LSE $M^{-1}s = r$.

Algorithm 1 PCG solver

procedure SOLVE(d)
  $\mathbf{p} \leftarrow \mathbf{0}$
  $\mathbf{r} \leftarrow \mathbf{d}$
  $\mathbf{s} \leftarrow$ `applyM`$(\mathbf{r})$
  $\rho \leftarrow \mathbf{r} \cdot \mathbf{s}$
  for $i \leftarrow 1 :$ `max_steps` **do**
    $\alpha \leftarrow \frac{\rho}{\mathbf{z} \cdot \mathbf{s}}$
    $\mathbf{z} \leftarrow$ `applyA`$(\mathbf{s})$
    $\mathbf{p} \leftarrow \mathbf{p} + \alpha \cdot \mathbf{s}$
    $\mathbf{r} \leftarrow \mathbf{r} - \alpha \cdot \mathbf{z}$
    if $\|\mathbf{r}\|_1 \leq$ `tol` **then**
      return $\mathbf{p}$
    end if
    $\mathbf{z} \leftarrow$ `applyM`$(\mathbf{r})$
    $\beta \leftarrow \frac{\mathbf{z} \cdot \mathbf{r}}{\rho}$
    $\rho \leftarrow \mathbf{z} \cdot \mathbf{r}$
    $\mathbf{s} \leftarrow \beta \cdot \mathbf{s} + \mathbf{z}$
  end for
end procedure

Incomplete Cholesky preconditioner. A is symmetric positive definite (spd), implying existence of a Cholesky factorization $A = LL^*$, where L is a lower triangular matrix. As the inverse of L is in general a dense matrix even for sparse A (rendering the computation infeasible), it is commonplace to use incomplete Cholesky (IC) factorization (so called because the sparsity pattern of A is retained during factorization) with any values outside the non-zero regions of A being discarded. The preconditioner M is given as the inverse of the IC factorization, and its diagonal entry $M_{i,j,k}$ can be computed according to

$$M_{i,j,k} = (A_{i,j,k} + M^2_{i-1,j,k} + M^2_{i,j-1,k} + M^2_{i,j,k-1})^{-1}. \tag{6}$$

2.2. Surface reconstruction

In FLIP simulation, the fluid surface at a particular timestep is defined as a level set at value 0 of a signed distance function (SDF). This SDF is determined by the positions of the particles (denoted by x_i):

$$\phi(\vec{x}_g) = ||\vec{x}_g - \bar{X}|| - \bar{r}, \tag{7}$$

where

$$\bar{X} = \frac{\sum_j k_1(||\vec{x}_g - \vec{x}_i||/R) x_j}{\sum_j k_1(||\vec{x}_g - \vec{x}_j||/R)}, \tag{8}$$

and $\bar{r}$ is a parameter representing the average particle radius. The kernel function k_1 is defined to decay with the distance between particle and grid point: $k_1 = \max(0, (1 - s^2)^3)$.

Notice that $\phi(\vec{x})$ is negative where $\vec{x}$ is in close proximity to many particles, i.e. within the fluid volume, and positive for $\vec{x}$ far away from the fluid. The isosurface of value 0 determines the fluid surface. This SDF is discretised by evaluating it at a set of points arranged in a regular grid. This discretisation is then used to compute a mesh approximation of the surface using the marching cubes algorithm [1]. The resulting meshes may be stored to disk and later imported into a rendering program in order to create realistic or stylized video footage of the simulation.

2.3. Cost analysis

Due to the complexity of the FLIP algorithm, we were forced to rely on instrumentation of the code to retrieve the cost of each substep. The number of floating point additions, multiplications, and divisions were measured alongside the number of memory reads, assuming a write-back/write-allocate cache. Note that only operations relevant for the algorithm were taken into account, while computations necessary for branch selection or index computations were discarded and deemed as overhead.

Substep	Additions	Multiplications	Divisions	Read bytes
Particle-to-grid	$2.50 \cdot 10^{10}$	$1.84 \cdot 10^{10}$	$2.51 \cdot 10^{6}$	$1.05 \cdot 10^{10}$
Pressure correction	$4.68 \cdot 10^{9}$	$4.35 \cdot 10^{9}$	$1.58 \cdot 10^{6}$	$9.47 \cdot 10^{10}$
Grid-to-particle	$6.16 \cdot 10^{8}$	$3.36 \cdot 10^{8}$	0	$2.84 \cdot 10^{9}$
Advection	$3.54 \cdot 10^{8}$	$2.24 \cdot 10^{8}$	3	$1.65 \cdot 10^{9}$

Table 1: Cost analysis for a "dam-break" simulation using $6\,272\,640$ particles on a $480\times160\times80$ grid.

For reference, the table above shows the operations count for a large simulation. Note that the number of divisions is several orders of magnitude less than other floating point operations, and is therefore neglected for the computation of performance and operational intensity.

Finally, it is interesting to point out that the disparity between bytes read in the pressure correction routine and other substeps will increase non-linearly with the problem size, thus values in Table 1 only provide a general idea.

3. OPTIMISATIONS

In the following we will outline the optimisations we performed on the different sections of the code. The optimisations are presented in the chronological order in which their implementation was completed.

3.1. Level set computation (v1.2)

Profiling of this substep showed that the external marching cubes algorithm used [2] contributed only marginally to the runtime. We therefore focused on the computation of the discretised level set. The original implementation already featured a set of approximations that allowed some optimisations. For example, each particle was given a finite region of influence in the computation of the level set function ϕ, reducing the complexity from $\mathcal{O}(m \cdot n)$ to $\mathcal{O}(n)$, where m is the number of grid corners and n is the number of points.

Many opportunities for optimisation were present, such as refactoring the weight function and the corresponding kernel function to avoid square roots, but also strength reduction, minimizing the use of conditionals inside loops, manual inlining, and separation of boundary and interior loops.

3.2. Particles data structure (v1.3)

Since FLIP is a hybrid Eulerian-Lagrangian method, it necessarily contains two major data structures: a grid to store fields, and an ensemble of particles to represent point-like samplings of the fluid volume and velocity. In principle, each particle can be represented by six floating-point numbers: a 3D position and a 3D velocity vector. In this section, we will cover the optimisation of the data structure representing the particle ensemble.

In the baseline implementation, individual particles were represented by a data structure, and the ensemble consisted of an array of these structures. This has two main performance-related issues.

Firstly, particle positions are alternated in memory with velocities (x-y-z-u-v-w-x...). In kernels where we only care about the particle's position, we end up indirectly loading some velocities anyway, just because they are in the same cache block as the particle's position. This effectively wastes bandwidth and lowers the kernel's computational intensity.

Secondly, the particle structures only allowed access to members through getters and setters. Since these were implemented in a separate `*.cpp` file, they became "black boxes" for the compiler, and could not be inlined. This hampered the compiler's ability to perform optimisations. Furthermore, in some instances the getters/setters were converting to/from Eigen vectors, which was unnecessary in most cases and introduced overhead.

In order to address all of the issues presented above, we opted to switch from an array of structs to a struct of arrays. This new particles data structure contains six arrays, one per component of a particle, which ensures memory contiguity of like data. Furthermore, direct access to these lists was allowed, thus removing the "black box" effect. With

no getter and setter methods, conversions to and from Eigen vectors were now only performed where strictly necessary (and were eventually removed entirely in subsequent optimisation passes).

3.3. Interpolation routine (v1.4)

In the context of FLIP, *interpolation* refers to the process of reconstructing the velocity vector at an arbitrary position in the simulation domain, from the MAC grid. Velocity interpolation is required by both the grid-to-particle projection and the particle advection steps.

In most cases, an interpolation call boils down to *trilinear* interpolation from eight values. However, if the position we want to interpolate to is in a boundary cell, we have two edge cases: if the point is near an outer face, we need to perform *bilinear* interpolation on that face, and if it is near an outer edge, we need to perform *linear* interpolation. Further, bilinear interpolation has six variants (one per face of a cube), and linear interpolation has eight (one per edge of a cube).

The situation is further complicated by the fact that, on a MAC grid, the three components of the velocity vector field are not stored in the same positions, but staggered on the different faces of the cells. This means the arrays for the different components have different dimensions ($(N_x + 1)\times N_y \times N_z$, $N_x \times (N_y + 1) \times N_z$, and $N_x \times N_y \times (N_z + 1)$) and different offsets from the origin. Finally, the MAC grid is actually storing *two* instances of the velocity field: $\vec{u}$ and $\vec{u}^*$, and because these are both needed to compute the velocity update to the particles by the FLIP algorithm, we need to be able to interpolate on one or both of these.

In a naive implementation, we might end up with a total of 90 different branches to determine what kind of interpolation we need to perform on what values. Since the actual interpolations are only a few flops, the runtime of an interpolation call is dominated by the logic required to determine what cell the given point is in, which kind of interpolation we should perform, and which data (array indices) to use.

In the baseline implementation, most of the branches were laid out manually, but inefficiently and interspersed with many function calls. The resulting code was over 600 lines long, and due to the large amount of code duplication, hard to optimise or maintain. We opted to rewrite the interpolation routine from scratch, adopting the following strategies to improve performance.

Automatically-inlined methods. The linear, bilinear and trilinear interpolation kernels were defined in header files and with the `inline` compiler hint. This allows us to avoid duplicating their code dozens of times, while allowing the compiler to inline them and maximize performance. To confirm that the compiler was indeed inlining the methods, we tried inlining them manually, and observed no change in performance.

Template argument for grid selection. The template argument `grid_name` was introduced to allow the caller to select which velocity component (u, v, or w) to interpolate on. Combined with `constexpr`, this allows us to perform a certain amount of the logic at compile time, e.g. to determine grid dimensions and offset, and significantly reduce the number of branches that need to be considered at runtime.

Template argument for multiple interpolation. In practice, we will either want to interpolate on just the velocity field $\vec{u}$ (as is the case in the advection substep), or on *both* $\vec{u}$ and $\vec{u}^*$ (as is the case in grid-to-particle projection). Since we know that $\vec{u}$ and $\vec{u}^*$ have the same dimensions and grid positions, we know that a given point will lie in the same cell for both grids. This means we can use the same indices for interpolating on $\vec{u}$ and $\vec{u}^*$, which will spare us significant overhead (recall that an interpolation call's runtime is dominated by index computations, not actual interpolation). By introducing a template argument `interpolation_mode`, we can again decide at compile time whether we want to interpolate on one or both grids, and coalesce two interpolation calls on $\vec{u}$ and $\vec{u}^*$ into one.

Conclusion. As outlined in Section 4, these optimisations greatly improved the performance of the particle advection and grid-to-particle projection substeps. The code complexity was also significantly reduced, with almost 300 fewer lines of code. Finally, we also considered vectorizing interpolation calls with SIMD instructions. However, given the high amounts of branching that still remains, vectorising the full routine would be difficult and of limited potential gain due to the low overall footprint of interpolation over the whole FLIP algorithm. An attempt was made at vectorizing just the trilinear interpolation kernel, however no performance gain was observed.

3.4. Particle-to-grid (v1.5)

For small and medium problem sizes, the particle-to-grid routine is comparable to the pressure correction substep in terms of runtime. Thus representing one of the main bottlenecks of the FLIP algorithm.

Manually inlined functions. The baseline implementation makes use of multiple functions to encapsulate the subroutines needed for the projection of particle velocities onto the staggered grid. Due to the velocity components being stored on cell interfaces the velocity fields u, v, and w have slightly different dimensions. For this reason, three versions for each subroutine were provided, each tailored to a different velocity field. In the optimised version, all the function calls were manually inlined, allowing to reduce code duplication, remove redundant computations, and in some cases enabling scalar replacement. Additionally inlining functions enabled fusion of some of the triple-nested loops present in the routine.

Boundary cells. To cope with the different dimensions of the velocity components u, v, and w, a large amount of branching and conditionals to avoid memory under- and overflows are needed, which imply overheads and may hinder compiler optimisations. However, the checks used in the loop responsible for the accumulation of velocities on cell interfaces were successfully moved outside the loop body, allowing for a more efficient treatment of inner cells. In other words, branching for boundary cells takes place outside the loop and a more efficient routine (i.e. free of unnecessary conditionals) is executed for particles contained in the interior cells of the computational domain.

In the end, these modifications enabled better optimisations from the compiler and were responsible for the main performance gains for the particle-to-grid substep.

Vectorization. At each timestep, the particle-to-grid routine iterates over each particle and retrieves the cell in which it is contained. Subsequently, it accumulates the weighted particle velocities onto a neighborhood of cell interfaces within a certain threshold. Unfortunately, at each timestep the order of the particles on the grid is unknown and (likely) different from the previous timestep due to the advection of particles. Moreover, the number of particles contained in each cell is not guaranteed to be constant. For these reasons, the accumulation loop which represents the most computationally intensive part of the particle-to-grid substep, is not prone to vectorization. However, the less intensive loop responsible for normalisation of the accumulated velocities was successfully vectorized using AVX2 intrinsics, resulting in a slight performance gain.

3.5. Pressure Correction (v1.6)

The bulk of the computational effort is due to solving the linear system of equations associated with the Poisson equation (5). The original implementation used the ICCG solver provided by the Eigen library [3]. In order to achieve speedup, we took took advantage of the specific characteristics of our problem by implementing our own ICCG solver. The algorithm is detailed, along with some possible avenues of optimisation, in [4].

Original: Sparse matrix construction. The original implementation rebuilt the matrix A for each timestep, which entailed iteratively adding triplets of the form `(row_idx, col_idx, value)` and then compiling the matrix into a Compressed Sparse Row (CSR) format. Each non-zero element of A is specified via a triplet using $2 \cdot s_{int} + s_{double} = 16B$, i.e. half of the memory usage is due to specifying the location of the entry and ultimately remains wasted due to the regular structure of the matrix. There are $6N_\mathrm{x}N_\mathrm{y}N_\mathrm{z} + \mathcal{O}(N_\mathrm{x}N_\mathrm{y} + N_\mathrm{x}N_\mathrm{z} + N_\mathrm{y}N_\mathrm{z}) \approx 6N_\mathrm{x}N_\mathrm{y}N_\mathrm{z}$ non-zero entries in A, where $N_\mathrm{x}, N_\mathrm{y}, N_\mathrm{z}$ denote the number of cells in x, y and z dimension, respectively. At a subsequent stage in constructing the sparse matrix, A will be converted to CSR format. Nonetheless, the triplets have to be written to memory first. The size of this triplet data structure can be bounded by $s_{A,\text{trip}} \leq N_\mathrm{x}N_\mathrm{y}N_\mathrm{z} \cdot 96B$. After conversion to CSR format, the matrix is stored in $s_{A,\text{CSR}} \leq N_\mathrm{x}N_\mathrm{y}N_\mathrm{z}(6s_{double} + 7s_{int}) = N_\mathrm{x}N_\mathrm{y}N_\mathrm{z} \cdot 76B$. Due to the way the creation of the matrix was implemented, specifying the values A_{ij} and A_{ji} together, there is necessarily an additional overhead in the Eigen method `SparseMatrix::setFromTriplets`, which in any case reads the list of triplets at least twice, according to the Eigen documentation.

Custom solver. Many of these performance issues could be effectively addressed by iterating directly over the CSR data structure and only changing the relevant entries. Even so, six out of the seven potentially non-zero entries (those corresponding to the neighboring cells) in each row are always either 0 or -1, resulting in a significant amount of memory being wasted by using double-precision floating points values to represent binary values.

It is clear that the generic solver provided by the Eigen library is severely limited in terms of performance on this restricted problem: as illustrated in Algorithm 1, a generic solver needs to be able to perform arbitrary matrix-vector products with sparse matrices, and – as explained – needs to store these matrices in a comparatively wasteful generic sparse matrix format. We therefore implement a custom ICCG solver in order to overcome these limits. Our solver is no longer generally applicable, but is now inseparable from the problem and even our concrete FLIP implementation.

Simplified Representation. By opting to store only the diagonal of the matrix explicitly, with all other values being computed at runtime, we are able to decrease the memory for storing A to one double per cell, or $s_{A,\text{diag}} = 8N_\mathrm{x}N_\mathrm{y}N_\mathrm{z}B$. The elements in the sub-diagonals and super-diagonals are inferred directly from the simulation data structures, using the cell index (to identify boundary cells) and an array indicating existence of fluid particles within each cell. This resulted in a significantly decreased use of memory, with an associated improvement in performance. Additionally, since the diagonal of A remains unaffected by the fluid, we can compute it during initialization and need not update it anymore.

Free-falling fluids. When unconstrained by boundary conditions (i.e. in free fall), the velocity field within the fluid is divergence-free. By using the solution to this trivial case, $\mathbf{p} \equiv 0$, as a starting point, situations with low, non-zero pressures are approximated within few conjugate gradient (CG) steps and trivial cases are solved immediately.

Blocking of `applyM`. The forward and backward substitution steps in the `applyM` subroutine shown in Algorithm 2 are seemingly strictly sequential, since each computation of an element of q depends on preceding elements. However, due to the three-dimensional nature of the problem, the boundary conditions break the dependencies between sub-

Algorithm 2 Application of preconditioner M

procedure APPLYM(r)
  for $i \leftarrow 1 : N_x, j \leftarrow 1 : N_y, k \leftarrow 1 : N_z$ **do**
    if cell (i,j,k) is fluid **then**
      $q_{i,j,k} \leftarrow M_{i,j,k} \cdot (r_{i,j,k} + M_{i-1,j,k} \cdot q_{i-1,j,k} + M_{i,j-1,k} \cdot q_{i,j-1,k} + M_{i,j,k-1} \cdot q_{i,j,k-1})$
    end if
  end for
  for $i \leftarrow N_x : 1, j \leftarrow N_y : 1, k \leftarrow N_z : 1$ **do**
    if cell (i,j,k) is fluid **then**
      $z_{i,j,k} \leftarrow M_{i,j,k} \cdot (q_{i,j,k} + M_{i,j,k} \cdot q_{i+1,j,k} + M_{i,j,k} \cdot q_{i,j+1,k} + M_{i,j,k} \cdot q_{i,j,k+1})$
    end if
  end for
  return z
end procedure

sequent cells that lie at boundaries. Therefore, several cells can be computed concurrently, as explained in [5], where the authors exploit this to implement a parallelised modified ICCG algorithm on a GPU.

Superscalar processors allow a similar approach to be taken to make use of instruction-level parallelism (ILP). This resulted in no speedup in our tests. We attribute this to the circumstance that our implementation is already memory-bound (c.f. Section 4). As the computation units of the CPU are not being exhausted, enabling more ILP is not expected to improve performance. On the other hand, this optimisation entailed decreasing the effective operational intensity by accessing the memory in a less structured manner, suggesting lower performance on an already memory-bound code.

3.6. Pressures AVX (v1.7)

Multiple calculations done in the pressures solver are slight modifications of the scalar product and hence really promising for vectorisation. The different calculations were packed into separate kernel functions which were optimised individually. In order to determine the optimal number of accumulators the following computations were conducted: the gap of Fused Multiply-Add (FMA) instructions on the Intel Skylake architecture is 0.5, meaning that 2 FMA instructions can be issued per cycle. In combination with a latency of 4 cycles we calculate the optimal number of accumulators to be 8. As each Advanced Vector Extensions (AVX) vector is able to hold 4 doubles, the loops were unrolled 32 fold. Some of the calculations involve returning the maximum value computed. This was done by increasing the number of specialized kernel functions to include versions that use combinations of other AVX instructions such as `_mm256_max_pd` and `_mm256_permute_pd` in order to find the maximum efficiently. Finally, all arrays that were passed to the kernels were memory aligned to further improve performance by using aligned loads.

4. EXPERIMENTAL RESULTS

In this section, the runtime improvements obtained by each optimisation step will be shown. Additionally, the performance behaviour of each substep of the FLIP algorithm will be studied with the aid of performance plots and the roofline model.

Experimental setup. As aforementioned, the target architecture of this project was Skylake. All benchmarks and tests were performed on an Intel Core i7-6700HQ CPU locked at 2.6GHz single-core (128KB L1, 1MB L2, 6MB L3).

For compilation the `gcc` compiler was used with the following flags: `-Ofast -march=native -mfma -std=c++17`. A variety of flags were tested to select this optimal combination.

Runtime. The histogram below shows the improvements in average runtime for a single FLIP step thanks to the different optimisations explained in Section 3. The contribution from each FLIP substep to the total runtime is highlighted with colors to help identifying the most time-consuming routines.

The timings shown in Fig. 1 were produced by benchmarking a "dam-break" simulation using 800 320 particles in a 240×80×40 grid.

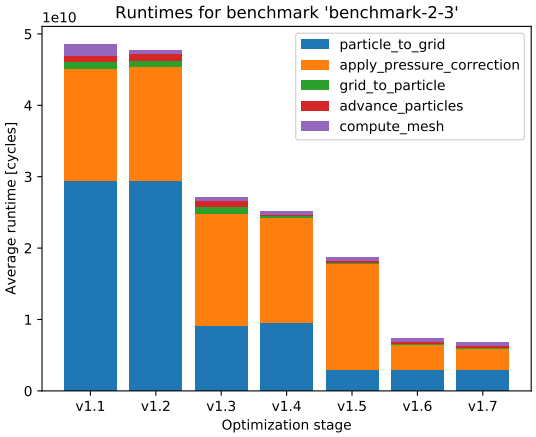


Fig. 1: Runtime (in cycles) of each optimisation stage. Note that some FLIP substeps are omitted due to their very small footprint.

Revised particles data structure (v1.3). This led to a more efficient use of the available bandwidth, better spatial locality, and enabled better optimisations from the compiler. The impact of these optimisations is visualized above (optimisation stage v1.3). Not all FLIP substeps are impacted

equally by changes to the particle representation; in particular, the pressure correction step does not use particles at all. The most significant improvement is in the particle-to-grid substep; for the benchmark case represented in Fig. 1, we observed a 3.2x speedup in the particle-to-grid step, and a 1.8x speedup overall. It is worth noting that no other optimisations were performed at this stage in the particle-to-grid step, only substitution of the particles data structure.

Rewrite of the interpolation routines (v1.4). The particle advection step had a speedup of 3.15x. The grid-to-particle step interpolates on both $\vec{u}$ and $\vec{u}^*$, so its observed speedup of 5.9x is almost twice as much, since it benefits from the multiple-interpolation template parameter.

Optimisation of the particle-to-grid routine (v1.5). Explained in Section 3.4, this yielded a further 3.1x speedup of the substep (i.e., adding this to the speedup achieved in v1.3 a total 9.9x speedup was achieved for the particle-to-grid substep) and a further 1.4x speedup overall. It is worth pointing out that the particle-to-grid substep is one of the main bottleneck of the algorithm, as shown in the baseline column (v1.1) in Fig. 1, thus its optimisation was of major importance and even little improvements had a positive tangible impact on the overall performance of the algorithm.

Pressure correction substep (v1.6 and v1.7). This substep is the main limiting factor in terms of performance of the algorithm, particularly for larger simulations, due to its high memory requirements and higher asymptotic complexity. We achieved a 4.9x speedup over the previous *Eigen*-based implementation, by exploiting knowledge of the underlying problem.

Overall a 7.6x speedup was achieved over the baseline implementation for the benchmark shown above. However, it is worth mentioning that for our largest simulation, using 6 272 640 particles, a higher speedup of 9.9x was achieved. This could be attributed to the improved scaling of the number of memory operations with increasing problem size in the optimised pressure solver, leading to greater differences in performance for larger simulations compared to the baseline.

Roofline. The roofline plot in Fig. 2 illustrates the performance achieved by the final optimised version of the FLIP algorithm and its difference to the peak performance of the machine, i.e. Skylake architecture supporting 2 FMA operations per cycle and 256-bit SIMD vectors.

It can be observed that the optimised pressure correction step is still memory bound, even though the memory requirements were greatly reduced as explained in Section 3.5. However, it is worth pointing out that the current implementation approaches the roofline for the measured bandwidth [6], meaning that further optimisations should focus on increasing the operational intensity of the routine. In the end, to achieve a better performance, a more memory efficient sparse linear solver would have to be developed to beat the current ICCG, which is already considered one of

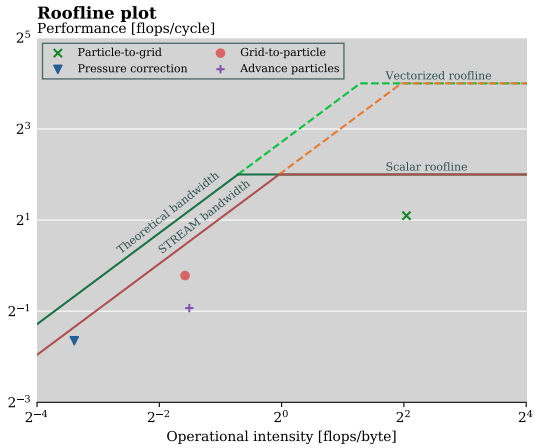


Fig. 2: Roofline plot with points for the most computational intensive FLIP substeps of the optimised version. Note that two different memory rooflines are shown: one for the theoretical bandwidth and a second one for the bandwidth measured by the STREAM benchmark.

the best choices for this specific problem [4].

On the other hand, the particle-to-grid routine lies on the compute bound side of the roofline plot, meaning it is less memory intensive. In this case, only half of the (scalar) peak performance was reached, probably due to the overhead caused by the large amount of conditionals required in the routine to avoid invalid memory accesses.

In the middle, but still on the memory bound side, lie the grid-to-particle and particle advection substeps, which are heavily dependent on the optimised velocity interpolation routines explained in Section 3.3, thus they are very similar in terms of operational intensity. Similarly to the particle-to-grid substep, for both grid-to-particle and advection the peak performance was not reached due to the high number of conditionals present in the interpolation routine to distinguish between internal cells and cells on the faces and edges of the cubical domain, where trilinear, bilinear and linear interpolation should be performed, respectively.

It is interesting to point out that the grid-to-particle substep achieves a better performance than the advection substep, even though the bottleneck of both is the interpolation routine. This difference is explained by the fusion of the interpolation for the velocities $\vec{u}$ and the intermediate velocities $\vec{u}^*$ in the grid-to-particle routine, which halves the amount of branching overhead.

In conclusion, these results show that little more performance could be gained in the pressure correction substep without changing the sparse linear solver.

Furthermore, the performance of the main components of the FLIP algorithm are limited by overhead due to branching inside loops. This could be mitigated by using a “ghost

cell" approach for the boundaries, at the cost of some additional memory usage.

Performance. The performance plot in Fig. 3 shows some quite interesting behaviours of the different substeps of the algorithm.

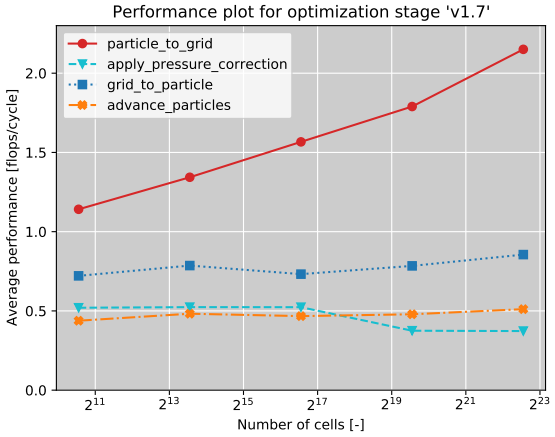
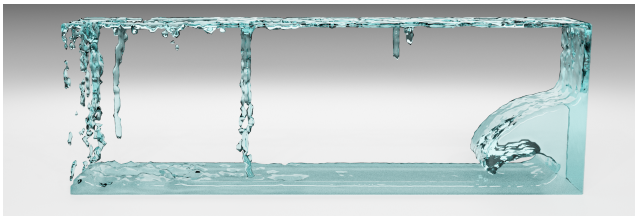


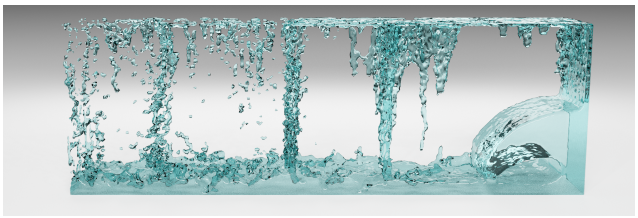
Fig. 3: Performance plot at optimisation stage v1.7 for the dominant FLIP substeps.

Firstly, it can be observed that the performance of the particle-to-grid substep increases with the increasing problem size. Normally, we would expect a flat performance line with drops corresponding to the L1, L2, and L3 caches being filled. This unusual behaviour could be explained by the different treatment of boundary and internal cells in the particle-to-grid routine, where the loop accumulating velocities for internal cells is free of unnecessary conditionals and more efficient than the one for boundary cells. With the increasing problem size, the ratio of particles contained in inner cells to those in boundary cells also increases, meaning that the larger the simulation the more times the efficient accumulation loop is executed compared to the slower loop for boundaries. For larger simulations (not visible in the plot, due to the high requirements in terms of memory and time of such a simulation) the performance of the routine will plateau to a constant value, either limited by the peak performance of the machine or, more likely, by the performance of the inner cells accumulation loop.

Looking at the performance of the pressure correction step, we notice a drop between 2^{17} and 2^{19} cells. This is due to the L3 cache being filled by the problem size. To verify this, let us estimate the maximum problem size that could fit into L3 cache (for the pressures substep). The pressure correction substep requires $3 \cdot 8$ B $= 24$ B per cell, since for each cell three doubles are required for the pressure correction routine: one entry of the diagonal of matrix A, one of the r.h.s. vector d and a last one for the pressures solution vector p. Based on this, we conclude that our 6MB cache



(a) Original implementation, 165,888 grid cells



(b) Optimised implementation, 768,000 grid cells

Fig. 4: Comparison of visual quality attainable in an equal amount of runtime, "dam-break" scenario.

can contain a problem size of about $\frac{6 \cdot 10^6 \text{ B}}{24 \text{ B}} \approx 2^{18}$ cells, which confirms our suspicions.

Finally, it is important to point out that a performance drop due to L3 cache overflow is only clearly visible in the pressure correction substep, due to its memory boundedness and its proximity to the peak performance roofline compared to the other kernels.

5. CONCLUSIONS

By performing single-core optimisations on a FLIP implementation, we were able to achieve speedups of 7.6x up to 9.9x on common problem sizes, while significantly reducing peak memory usage, enabling much larger-scale simulations on existing hardware. Using efficient data structures, we were able to achieve performance within a factor of two of the hardware performance bounds. Using template arguments to specialise the interpolation routines to each grid type, thus enabling additional compiler optimisations, resulted in substantially reduced runtime overhead due to index computations. By streamlining the inner loops of the particle-to-grid step, we obtained 3.1x speedup on this substep. By implementing an optimised and vectorized ICCG solver tailored to our problem structure, we were able to outperform the high-performance general-purpose implementation provided by the Eigen library by a factor of 4.9x. Our optimised implementation allows for simulations of much higher quality in comparable time, as illustrated in Fig. 4.

6. CONTRIBUTIONS OF TEAM MEMBERS (MANDATORY)

Christoph. Timing methods. Level-set method optimisations (see Section 3.1, equal contributor). CG pressure solver (see Section 3.5; implementation, scalar optimisations). Non-beneficial optimisations attempted: blocking of forward & backward substitution in pressure solver.

Sean. Design and implementation of new particles data structure (see Section 3.2). Rewrite of the interpolation routines (see section 3.3). Work on benchmarking: designing, running and plotting. Attempted some additional optimisations: particle sorting, particle cell index caching.

Felix. Level set optimisations (see Section 3.1,equal contributor), CG Pressure solver (general outline and vectorisation)

Mikael. Integrated the new particles data structure in the FLIP methods. General optimisations (scalar replacement, strength reduction, replaced getters and setters with direct access, removed Eigen data structures) on all FLIP methods except the pressure correction and surface reconstruction substeps. optimised the particle-to-grid routine (see Section 3.4) and less intensive substeps (application of external forces and boundary conditions). Cost analysis and roofline plot. Validation system using netCDF.

7. REFERENCES

[1] William E. Lorensen and Harvey E. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," 1987.

[2] Alec Jacobson, Daniele Panozzo, et al., "libigl: A simple C++ geometry processing library," 2018, https://libigl.github.io/.

[3] Gaël Guennebaud, Benoît Jacob, et al., "Eigen v3," http://eigen.tuxfamily.org, 2010.

[4] R. Bridson and M. Müller-Fischer, "Fluid simulation: Siggraph 2007 course notes," in *SIGGRAPH '07*, 2007.

[5] Jiaquan Gao, Bo Li, and Guixia He, "Modified incomplete cholesky preconditioned conjugate gradient algorithm on gpu for the 3d parabolic equation," in *Network and Parallel Computing*, Ching-Hsien Hsu, Xiaoming Li, Xuanhua Shi, and Ran Zheng, Eds., Berlin, Heidelberg, 2013, pp. 298–307, Springer Berlin Heidelberg.

[6] J. D. McCalpin, "STREAM Benchmark," https://www.cs.virginia.edu/stream/, 2013.