UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

GPU-Beschleunigung der Material Point Method

Masterarbeit

zur Erlangung des Grades Master of Science (M.Sc.)
im Studiengang Computervisualistik

vorgelegt von

Fabian Meyer

Erstgutachter:	Prof. Dr.-Ing. Stefan Müller (Institut für Computervisualistik, AG Computergraphik)
Zweitgutachter:	Bastian Krayer, M.Sc. (Institut für Computervisualistik, AG Computergraphik)

Koblenz, im Januar 2019

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Abstract

Physic

Contents

1 Introduction

2 Related Work

Particle-in-Cell (PIC): PIC techniques were developed initially for hydrodynamics [EHB57]. They are also widely used in plasma simulations to solve Maxwell's equations. Their key steps are identical to the MPM-method:

1. Transfer particle data to a grid.
2. Solve the governing partial differential equations on the grid to move forward in time.
3. Transfer grid data back to particles.

In general the grid is regarded as a scratch-pad since its only a temporary storage for the means of computing item 2. Based on PIC to reduce dampening and momentum conservation Fluid-Implicit-Particle (FLIP) was developed for fluid simulations and is still widely in use for large simulations. Their counterpart being the empirical Smoothed Particle Hydrodynamics (SPH) [GM77].

For that matter APIC was developed as an improvement to PIC and FLIP solvers. Due to the definition of a local affine velocity field around each particle (instead of a single vector) APIC conserves angular momentum, reduces damping and improves stability over PIC and FLIP, more in chapter 4.5.3. [JST17] [Jia+15]

A step further goes PolyPIC giving the velocity field even more freedom by adding polynomial modes of increasing order to the description. [Fu+17]

Material Point Method: The MPM-Method is like the PIC-Method mathematical and physical in that it uses analysis and numerics to be derived without the need for empirical assumptions. Originally it was developed by [SZS95] with a focus on the dynamics of solids. [Sto+13] brought the method into computer graphics for their animation film *Frozen*. In rapid succession a variety materials and effects got modeled by the MPM-Method. Since MPM is based on discretizing governing equations and using constitutive models covered by the different literature its fast rate of expenditure is not a surprise.

The simulation of (hyper-)elastic material can be done with a number of different models utilizing fixed Linear strain 4.3.2, left Cauchy-Green strain 4.2 or Hencky-strain(logarithmic) [Guo+18] for the different requirements of small to large strains. Inserting these into the different energy density $\Psi(\epsilon)$ models can already result in a constitutive model. Plasticity in general requires extra modeling in terms of a flow rule, yield-condition and a hardening rule which are not discussed here [Öch14]. To give a short impression of the developments in computer graphics:

[Sto+14] model phase transition (melting) to a liquid with the fixed corotational hyperelasticity (4.3.2) and discretizing the heat equation. [Ram+15] use an Olroyd-B model with Cauchy-Green strain to handle elasticity and plasticity for

viscoelastic fluids, foams and sponges. [Klá+16] uses Hencky-strain in combination with a St. Venant-Kirchhoff model and Drucker-Prager elastoplascity to model sand. [Tam+17] combine particles of different species (sand and water) by one grid for each species and relating them with a heuristic momentum exchange. [Guo+18] simulates thin shells (cloths,cups etc.) with a Kirchhoff-Love model using Hencky-strains.

There exist several implementations utilizing the GPU for Particle-in-Cell type techniques, which the Material Point Method is a part of:

Particle sorting: [Pay12] studies different particle list sorting methods. Methods like 'Message Passing Sort' [Kon+11] [DS11], 'In Place Particle-QuickSort' [SDG08] and 'Linked List Ordering' [Bur+10] try to utilize the partial sorting and only update particles that need to be moved within the list. These methods often rely on the assumption that only a small number of particles change their grid node, where in contrast MPM can be quite dynamic. Those that do move, move at most to one neighboring grid-node or stay in the same node due to the CFL-condition, chapter 4.5.4. These methods often use fixed bin-sizes which can be quite memory intensive and add additional memory management. Fixed size binning make the methods mentioned before easier to implement, which don't necessarily map well to the GPU due to their high complexity and uncoalesced accesses. As a result [Pay12] come to the conclusion to fall back to the CUDA library `thrust`, although they do note that the aforementioned methods can perform faster for particles that move only a limited amount of grid nodes. `thrust` provides the radix sort method `thrust::sort_by_key()`, where one sorts by grid index as key.

In general full or index sorts can be done. A full sort reorders every variable of a particle. An index sort gives back an index with an offset to access the particle variable.

[Hoe14] compare radix to counting sort utilizing a uniform grid and use the latter for SPH-fluids.

Particle-to-Grid=Transfers: The particle to grid transfer is generally done in two manners [SDG08]:

1. **Particle Push:** X particles in a local neighborhood of a grid point push their attributes onto it. Since this relationship results in a race condition `atomicAdd(float f)`s or explicit synchronization is necessary, e.g. atomic: [Pay12], synchronized: [Ros13].

2. **Particle Pull:** A grid point pulls the particles in its local neighborhood from a sorted particle list. This has the benefit of no race conditions being present.

A Grid-to-Particle-Transfer is in general easier since a particle already knows, due to its position, the grid points in its local neighborhood. The grid points in a uniform grid are 'sorted' by nature.

Generally these methods are augmented by splitting the grid in 2D/3D blocks making use of the shared memory structure of the GPU [Pay12] [Hön+] [Klá+17],

often called domain decomposition. Shared memory is however limited per GPU (GTX 970, 48KB per block).The grid nodes reached by the support of the interpolation function on each side also have to be loaded into memory, typically called the halo of a block. Since the number of particles per grid point in the MPM-method is generally between 4-10 only very limited block sizes are available [Sto+13].

In Particle Push methods partial sums are also computed on the halo. Two different techniques exist to get the data between blocks coherent again:

1. Atomically add up the values of the halo. This results in two atomic adds on a side node, four on an edge node and eight on a corner-node; If the block extend is bigger than half the halo extend.

2. Have every halo of every block correspond to actual global memory and do add steps along each of the three axes. This is infeasible for large supports due to the added memory requirements. [Cra+11]

Stencil computations and filtering techniques underlie a very similar process to MPM-transfers. Although these techniques employ a one-to-one grid relationship, since no particles are involved, it is worth taking a look at them.

Widely used is a 2.5D blocking: A domain (of the domain decomposition) is split into 2D-planes along one axis. Data reuse of shared memory is maximized by a cyclic-queue strategy: Start a thread for each grid node in the first plane and load the halo along the axis into shared memory. Then each thread iterates along the axis over the 2D-planes synchronously, discarding the oldest plane in memory while loading the next plane (along the axis) into shared memory. Thus there are no z-dimension halos. [BP10] [Wil+07] [KD13]

An interesting prospect for filtering techniques on the GPU is given by NVIDIA's new shuffle operations. Shuffle allows to 'share' register memory between threads of a warp [Wes15]. They are however not supported on other architecture, yet.

Particle Activation: If a particle is under a user-defined velocity threshold a particle becomes deactivated. It does not get recognized by any particle to grid transfer (nor any grid transfer not affected by the velocity transfer). They may of course be reactivated if the back transfer of the velocity exceeds that threshold again or they collide. [Klá+17]

Dividing the particles up can be realized with stream compaction algorithms: One implementation would be to compute a scan (based on the discussed condition) followed up by a modified reorder shader discussed in chapter 5. The modified reorder shader would then be coalesced.

Particle Resampling: A method that could alleviate pressure off fixed bin sizes and could make the partial sorting methods as well as the shared-memory restrictions less of a factor is particle resampling. Their intention mostly is to fill numerical gaps in the material at large deformations. Split and merge methods can directly control how many particles in a cell are allowed and accordingly increase or reduce them.

A simple *Split* method divides one particle into eight with a distance of half the diagonal length of a grid cell from the original particle. The eight particles span a cube that can be randomly rotated if desired. Mass and volume get equally distributed; Velocity and deformation gradient get copied. Refer to chapter 4.5 for conservation of mass and momentum in MPM. [Gao+17]

A *Merge* method looks for the closest neighbors and creates the new merged particle at the geometric center. Mass and volume are accumulated. Velocity is computed from a mass-weighted average of the participating particles. The deformation gradient is accordingly to chapter 4.2 decomposed into $\boldsymbol{F} = \boldsymbol{U}^T$. The quaternion average of $\boldsymbol{U}$ and $\boldsymbol{V}$ leads to an average rotation $\overline{\boldsymbol{U}}$ and $\overline{\boldsymbol{V}}$, respectively. The principal strains $\boldsymbol{\Sigma}$ are just averaged component-wise $\overline{\boldsymbol{\Sigma}}$. One can compose them in the same fashion $\overline{\boldsymbol{F}} = \overline{\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}}^T$.[Gao+17]

Unconditionally keeping particles on a fixed size is not desirable due to fracture requiring that gaps will be created. [Gao+17] produce a signed distance field to identify interior points. Since fracture typically starts at a surface point desired behavior is accomplished again. Furthermore the signed distance field can be used to adapt the grid size depending on the distance to the surface. This allows coarser levels with increasing distance from the surface.

3 Notation

Multi-component types are generally printed in bold letters $\boldsymbol{A}$, $\boldsymbol{a}$. Vectors furthermore use lower-case letters $\boldsymbol{v}$, while Matrices use upper-case letters $\boldsymbol{M}$. A variable in the Lagrangian form has a subscript 0 ${}_0\boldsymbol{x}$, the Eulerian description will be written with an 'overline' ${}_t\boldsymbol{x}$ if distinction is needed. Occasionally notation may be omitted where it is apparent from context.

Beginning from chapter 4.5 variables defined on a particle will have subscript p (e.g. $\boldsymbol{x}_p$). Grid cells will be assigned a subscript i or additionally j denoting the grid index (e.g. $\boldsymbol{x}_i$). This is not to be confused with the following Einstein-Notation. Therefore beginning with chapter 4.5 Einstein-Notation for components will use greek letters (α, β). Occasionally summation over grid index i and particles p may be implied. A variable of the n-th time step with associated time $t^n = \sum_{i=1}^{n} \Delta t^i$ will have a right superscript: $\boldsymbol{x}^n$.

3.1 Einstein-Notation

Used throughout this thesis is at instances the Einstein-Notation when using tensors and vectors. Let $\boldsymbol{a}$ and $\boldsymbol{b}$ be vectors of dimension n and $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{D}$ $m \times n$ tensors. Repeated indices that are also defined on the variable imply component-wise operations. $a_i b_j$ is multiplying component i with component j of vectors $\boldsymbol{a}$ and $\boldsymbol{b}$. Vector and tensor/matrix addition thus become:

$$c_i = a_i + b_i \quad \text{and} \quad D_{ij} = A_{ij} + B_{ij}. \tag{1}$$

A transpose operator swaps the indices:

$$D^T_{ij} = D_{ji}. \tag{2}$$

Repeated indices that are not otherwise defined however imply summation on that index. The vector dot product becomes:

$$a_i b_i \equiv \sum_{i=1}^{n} a_i b_i. \tag{3}$$

Following this notation the Frobenius inner product between two second-order tensors is (also called Frobenius scalar product):

$$\boldsymbol{A} : \boldsymbol{B} \equiv A_{ij} B_{ij} \equiv \sum_{i=1}^{n} \sum_{j=1}^{m} A_{ij} B_{ij}. \tag{4}$$

The Frobenius inner product between a $r \times s \times m \times n$ fourth-order tensor $\boldsymbol{C}$ and second-order tensor creates a second order tensor combining the ideas of 1 and 4:

$$\boldsymbol{A} = A_{ij} = \boldsymbol{C} : \boldsymbol{B} = C_{ijkl} B_{kl} = \sum_{k=1}^{m} \sum_{l=1}^{n} C_{ijkl} B_{kl}. \tag{5}$$

Matrix-Vector and Matrix-Matrix multiplication can be expressed this way as:

$$b_i = A_{ij}a_j \quad D_{ij} = A_{ik}B_{kj}. \tag{6}$$

The definition of the Kronecker Delta is as follows:

$$\delta_{ij} = \begin{cases} 1 & \text{if} \quad i = j \\ 0 & \text{if} \quad i \neq j \end{cases} \tag{7}$$

The Kronecker Delta as an operator is most efficiently described as a substitution:

$$a_i\delta_{ij} = a_j. \tag{8}$$

I.e. this component is only evaluated if $i = j$. In this case it becomes equivalent to the identity matrix $\boldsymbol{I}$. The Kronecker Delta comes up in differentiating a variable by itself with different indexing:

$$\frac{\partial x_0}{\partial x_1} = 0, \frac{\partial x_0}{\partial x_0} = 1 \Rightarrow \frac{\partial x_i}{\partial x_j} = \delta_{ij}. \tag{9}$$

The alternating tensor or Levi-Civita symbol

$$\varepsilon_{ijk} = \begin{cases} +1 & \text{if } (i,j,k) \text{ is } (1,2,3), (2,3,1), \text{ or } (3,1,2) \\ -1 & \text{if } (i,j,k) \text{ is } (3,2,1), (1,3,2), \text{ or } (2,1,3) \\ 0 & \text{if } i = j, \text{ or } j = k, \text{ or } k = i \end{cases} \tag{10}$$

is used to express cross-products $\boldsymbol{c} = \boldsymbol{a} \times \boldsymbol{b}$:

$$c_i = \varepsilon_{ijk}a_jb_k \left(= \sum_{j=1}^{n}\sum_{k=1}^{n} \varepsilon_{ijk}a_jb_k \right). \tag{11}$$

A useful relation to the Kronecker-Delta is:

$$\varepsilon_{ijk}\varepsilon_{imn} = \delta_{jm}\delta_{kn} - \delta_{jn}\delta_{km}. \tag{12}$$

[McG]

4 Basics

4.1 Reference and current configuration

Particles (or material points) in continuum mechanics are not what classically might be thought of as a particle. Rather the continuum assumption holds: each particle represents a continuous piece of material, s.t. a microscopic view does not need to be adopted. A particular body is composed of a set of particles and can adapt different configurations due to changes in shape. These changes are caused by external or internal effects (forces etc.) on it and deform the body over time. [Abe12] [Jia+16]

When modelling solids changes of quantities from an initial reference configuration ${}_0\boldsymbol{x}$ to another current configuration ${}_t\boldsymbol{x}$ need to be measured. In the material point method the reference configuration ${}^0\boldsymbol{x}$ is just the initial configuration of the body (at time $t = 0$). This is similar to the total Lagrangian formulation in finite element methods [Bat06].

Let $\Omega^0, \Omega^t \subset \mathbb{R}^3$ be the set of (material) points in the reference and current configuration respectively. Then one may define a function or mapping ${}^t_0\phi(\cdot, t) : \Omega^0 \to \Omega^t$, which relates the reference configuration to the current configuration. If we assume for simplicity that ${}_0\boldsymbol{x}$ and ${}_t\boldsymbol{x}$ describe the position of the particle in their respective configurations, this mapping becomes the deformation of the body from the reference configuration ${}_0\boldsymbol{x}$:

$$ {}_t\boldsymbol{x} = {}^t_0\phi({}_0\boldsymbol{x}, \boldsymbol{t}) \tag{13} $$

If for instance the body consisting of each material point ${}_t\boldsymbol{x}$ moves with velocity $\boldsymbol{v}$ and rotation $\boldsymbol{R}(t)$, we can defines this mapping to be:

$$ {}_t\boldsymbol{x} = {}^t_0\phi({}_0\boldsymbol{x}, \boldsymbol{t}) = \boldsymbol{R}(t){}_0\boldsymbol{x} + \boldsymbol{v}t \tag{14} $$

The velocity of a material point in ${}_0\boldsymbol{x}$ can be defined using this mapping

$$ {}_0\boldsymbol{v}({}_0\boldsymbol{x}, t) = \frac{\partial^t_0\phi}{\partial t}({}_0\boldsymbol{x}, t) \tag{15} $$

and similarly the acceleration is defined

$$ {}_0\boldsymbol{a}({}_0\boldsymbol{x}, t) = \frac{\partial^{2t}{}_0\phi}{\partial t^2}({}_0\boldsymbol{x}, t) = \frac{\partial_0\boldsymbol{v}}{\partial t}({}_0\boldsymbol{x}, t). \tag{16} $$

It is helpful to abstract away from the reference configuration and think of it as being defined in a different fixed material space. Physically this has the impact of moving with the particle in world space. Commonly known as the Lagrangian form. It is often easier in continuum mechanics to start with a Lagrangian description and switch to a Eulerian one if needed. The Eulerian description is static: Variables of particles that are moving by are measured while staying in a fixed position.

These descriptions are different but they will yield the same measurements when related correctly. I.e. both configuration refer variables defined on them to the deformed state, but the position where the 'look up' of that value happens is different. In the reference configuration we look up at the initial position of the particle. In the current configuration we look up at the particle's 'world position'. These relations for some particle quantity f are called the (Lagrangian) pull back

$$ {}_0f({}_0\boldsymbol{x}, t) = {}_tf({}^t_0\phi({}_0\boldsymbol{x}, t), t) \tag{17} $$

and the (Eulerian) push forward

$$ {}_tf({}_t\boldsymbol{x}, t) = {}_0f({}^t_0\phi^{-1}({}_t\boldsymbol{x}, t), t) = {}_0f({}^0_t\phi({}_t\boldsymbol{x}, t), t) \tag{18} $$

with definitions over their respective spaces ${}_tf(\cdot, t) : \Omega^t \to \mathbb{R}$, ${}_0f(\cdot, t) : \Omega^0 \to \mathbb{R}$. To enable the operator ${}^t_0\phi$ to be homeomorphic, s.t. an inverse ${}^t_0\phi^{-1} = {}^0_t\phi$ is defined, it is assumed that no two particles will ever occupy the same space at the same time.

The difficulty in the eulerian formulation becomes apparent when differentiating (due to the chain rule):

$$ \frac{\partial}{\partial t}{}_0f_i({}_0\boldsymbol{x}, t) = \frac{\partial_t f_i}{\partial t}({}^t_0\phi({}_0\boldsymbol{x}, t), t) + \frac{\partial_t f_i}{\partial_t x_j}({}^t_0\phi({}_0\boldsymbol{x}, t), t)\frac{\partial^t_0\phi_j}{\partial t}({}_0\boldsymbol{x}, t) \tag{19} $$

Combining this with equation 15 and applying the push forward 18 to cancel out mappings leads to the definition referred to as the material derivative:

$$ \frac{D}{Dt}f({}_t\boldsymbol{x}, t) = \frac{\partial f}{\partial t}({}_t\boldsymbol{x}, t) + \frac{\partial f}{\partial x_j}({}_t\boldsymbol{x}, t)v_j({}_t\boldsymbol{x}, t) \tag{20} $$

The Jacobian of the deformation map ϕ is the deformation gradient $\boldsymbol{F}$ and is one of the key components to measure strain:

$$ {}^t_0F_{ij}({}_0\boldsymbol{x}, t) = \frac{\partial^t_0\phi_i}{\partial_0 x_j}({}_0\boldsymbol{x}, t) = \frac{\partial_t x_i}{\partial_0 x_j}({}_0\boldsymbol{x}, t) \tag{21} $$

Intuitively it measures the local deformation of all particles in a small neighborhood of ${}_0B_\epsilon$ to ${}_tB_\epsilon$. Topology specifies the neighborhood using the open ball concept ${}_{0,t}B_\epsilon(\boldsymbol{x}) = \{\boldsymbol{y} \in \Omega^{0,t} | d(\boldsymbol{x}, \boldsymbol{y}) < \epsilon\}$ given a distance measure d. ${}_0B_\epsilon$ becomes the pre-image of ${}_tB_\epsilon$ under ${}^t_0\phi$. This allows to describe infinitesimal changes in position from the reference to the current configuration

$$ d_t x_i = {}^t_0F_{ij}d_0x_j. \tag{22} $$

With this quantity in place volume and area changes are calculable. In a typical analytical fashion a coordinate system change ${}_0x \to {}_tx$ is done using the Jacobian

(determinant) matrix. The determinant is given a separate name ${}^t_0J = \det({}^t_0\boldsymbol{F})$. The push forward of ${}_0g : \Omega^0 \to \mathbb{R}^d$ is ${}_tg : \Omega^t \to \mathbb{R}^d$:

$$\int_{{}_t\boldsymbol{B}} {}_tg({}_t\boldsymbol{x}, t) d_t\boldsymbol{x} = \int_{{}_0\boldsymbol{B}} {}_0g({}_0\boldsymbol{x}, t) {}^t_0J d_0\boldsymbol{x}. \tag{23}$$

This can also be achieved by the cross product. A cube spanned by vectors ${}_0\boldsymbol{x}_i$ ($i = 1, 2, 3$) becomes a parallelepiped in the deformed configuration $d_tV = |{}^t_0\boldsymbol{F} d_0\boldsymbol{x}_0 \cdot ({}^t_0\boldsymbol{F} d_0\boldsymbol{x}_1 \times {}^t_0\boldsymbol{F} d_0\boldsymbol{x}_2)| d_0V$:

$$d_tV = {}^t_0J d_0V. \tag{24}$$

The area change is given by Nanson's Formula. Where $d\boldsymbol{A}$ is a vector pointing in the direction of the normal of the area:

$$\begin{aligned} & d_tV = d_t\boldsymbol{A} \cdot d_t\boldsymbol{l}, \quad d_0V = d_0\boldsymbol{A} \cdot d_0\boldsymbol{l} \\ & \overset{22,24}{\Longrightarrow} {}^t_0J d_0\boldsymbol{A} \cdot d_0\boldsymbol{l} = d_t\boldsymbol{A} \cdot ({}^t_0\boldsymbol{F} d_0\boldsymbol{l}) \\ & \Rightarrow d_t\boldsymbol{A} = {}^t_0\boldsymbol{F}^{-T} {}^t_0J d_0\boldsymbol{A} = {}^0_t\boldsymbol{F}^T {}^t_0J d_0\boldsymbol{A}. \end{aligned} \tag{25}$$

A surface integral may then be transformed to reference configuration by

$$\int_{\partial_0 B} \boldsymbol{h}({}_t\boldsymbol{x}, t) \cdot d\boldsymbol{A}({}_t\boldsymbol{x}) = \int_{\partial_t B} \boldsymbol{h}({}_0\boldsymbol{x}, t) \cdot \boldsymbol{F}^{-T} J d\boldsymbol{A}({}_0\boldsymbol{x}) \tag{26}$$

where ${}_0\boldsymbol{h} = \boldsymbol{h}({}_0\boldsymbol{x}, \cdot)$ is the pull back of ${}_t\boldsymbol{h} = \boldsymbol{h}({}_t\boldsymbol{x}, \cdot)$. $d_0\boldsymbol{A}$,$d_t\boldsymbol{A}$ point in the direction of the surface normal of $\partial_0 B({}_0\boldsymbol{x})$, $\partial_t B({}_t\boldsymbol{x})$, respectively. [Abe12] [Jia+16]

4.2 Polar- and Singular Value Decomposition

The target is to define strain measures in terms of the deformation gradient: $\boldsymbol{\epsilon}(\boldsymbol{F})$. In equation 14 a rigid body movement was introduced. A problem arises when calculating the deformation gradient of this equation. Let $b_i(t) = v_i t$ be more generally a translation:

$${}^t_0F_{ij} = \frac{\partial_t x_i}{\partial_0 x_j} = \frac{\partial(R_{ik}(t) {}_0x_k + b_i(t))}{\partial_0 x_j} = R_{ik}(t)\delta_{kj} = R_{ij}(t) \tag{27}$$

As can be seen the deformation gradient contains a rigid rotation. For strain measures this is not beneficial as an assumption of the stiffness tensor requires no net-rotation (more in chapter 4.3.1). I.e. the deformation gradient has two components a constant rotation and the actual distortion or strain. There is two ways to deal with this:

1. Use a strain measure that cancels out the rotation. An example for this would be the Green-strain with quadratic components:

$$E_{ij} = \frac{1}{2}\left(F_{ki}F_{kj} - \delta_{ij}\right). \tag{28}$$

2. Polar decompose the deformation gradient in its rotational $\boldsymbol{R}$ and (symmetric positive definite) distortion $\boldsymbol{S}$ part $\boldsymbol{F} = \boldsymbol{RS}$.

That equation 28 cancels out the rotational part can be shown by item 2:

$$\frac{1}{2}\left(F_{ki}F_{kj} - \delta_{ij}\right) = \frac{1}{2}\left(S_{mi}R_{km}R_{kn}S_{nj} - \delta_{ij}\right)$$

$$= \frac{1}{2}\left(S_{mi}\delta_{mn}S_{nj} - \delta_{ij}\right) = \frac{1}{2}\left(S_{im}S_{mj} - \delta_{ij}\right)$$

$$= \frac{1}{2}\left(\boldsymbol{S}^2 - I\right).$$

Regarding item 2: Assuming a singular value decomposition of

$$\boldsymbol{F} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T \tag{29}$$

is already computed, where $\boldsymbol{U}$,$\boldsymbol{V}$ are orthogonal matrices and $\boldsymbol{\Sigma}$ is a diagonal matrix containing the singular values $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_r$ of $\boldsymbol{F}$. $(r - n)$ forms the dimension of the null space. The polar decomposition is computable as:

$$\boldsymbol{R} = \boldsymbol{V}\boldsymbol{W}^T, \quad \boldsymbol{S} = \boldsymbol{W}\boldsymbol{\Sigma}\boldsymbol{W}^T. \tag{30}$$

Since singular values are positive, it is straightforward to see that the properties for $\boldsymbol{R}$ and $\boldsymbol{S}$ hold. The components of the singular value decomposition are important to gain an intuition for its usefulness: The columns of $\boldsymbol{U}$ and $\boldsymbol{V}$ span bases for the row and column spaces of $\boldsymbol{F}$ using the left and right singular vectors $\boldsymbol{u}_i, \boldsymbol{v}_i$, respectively [MIT]. For illustrating purposes imagine the manipulation of $\boldsymbol{v}_1$ due to $\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$:

1. Transform from the right singular vector space to standard basis space: $\boldsymbol{V}^T\boldsymbol{v}_1 = \boldsymbol{e}_1$
2. Scale by singular values to transform to principal stretch space: $\boldsymbol{\Sigma}\boldsymbol{e}_1 = \sigma_1\boldsymbol{e}_1$
3. Transform to left singular vector space: $\boldsymbol{U}\sigma_1\boldsymbol{e}_1 = \sigma_1\boldsymbol{u}_1$

For a positive definite matrix the singular value decomposition becomes even easier as $\boldsymbol{U} = \boldsymbol{V}$. Item 3 then effectively just becomes a transform 'back'.

In the following a summary of the 3×3 singular value decomposition as in [McA+11a] is given. The proposed singular value decomposition is also called the 'Polar SVD' and follows a specific convention.

1. $\boldsymbol{U}$,$\boldsymbol{V}$ are reflection-free corresponding to true rotation matrices, i.e. both $\det(\boldsymbol{U}), \det(\boldsymbol{V}) = 1$ hold.
2. If $\det(\boldsymbol{F}) = -1$ the negative sign needs to move on to $\boldsymbol{\Sigma}$ as a result of item 1. The lowest singular value in magnitude will get a negative sign attached.

This convention does not change the existence or uniqueness of the singular value or polar decomposition, although strictly speaking $\boldsymbol{S}$ in the Polar decomposition is not positive definite anymore.
The algorithm proceeds as follows:

1. **Symmetric eigenanalysis:** A Jacobi eigenvalue algorithm begins with the symmetric positive definite matrix $\boldsymbol{S}^{(0)} = \boldsymbol{A}^T\boldsymbol{A} = \boldsymbol{V}\boldsymbol{\Sigma}^2\boldsymbol{V}^T$.

   1.1. Iteratively compute (an also symmetric, positive definite) $\boldsymbol{S}^{(k+1)} = [\boldsymbol{Q}^{(k)}]^T\boldsymbol{S}^{(k)}\boldsymbol{Q}^{(k)}$, where $\boldsymbol{Q}$ is a Givens rotation aiming to eliminate off S_{12}. Store $\boldsymbol{V}^{(k+1)} = \boldsymbol{V}^{(k)}\boldsymbol{Q}^{(k)}$.

   1.2. Do 1.1. again for the other off-diagonal entries S_{13}, S_{23}.

   1.3. Redo 1.1. - 1.2. a fixed amount of steps m.

2. **Sorting singular values:** Compute $\boldsymbol{B} := \boldsymbol{A}\boldsymbol{V}$, where $\boldsymbol{V} = \boldsymbol{V}^{(3m)}$. Acquire $\boldsymbol{\Sigma}$ by $\|\boldsymbol{b}_i\|_2 = \|\boldsymbol{u}_i\sigma_i\|_2 = |\sigma_i|$, where also $\boldsymbol{B} = \boldsymbol{U}\boldsymbol{\Sigma}$ holds. Permute the singular values by sorting them in decreasing order. Apply the same permutation to the columns of $\boldsymbol{B}$ and $\boldsymbol{V}$, where switches in $\boldsymbol{V}$ also cause a sign change. Enforce the convention mentioned above.

3. **QR-Factorization:** Compute $\boldsymbol{U}$ using a $\boldsymbol{QR}$-Factorization with Givens Rotations, where $\boldsymbol{B} = \boldsymbol{Q}\boldsymbol{R} = \boldsymbol{U}\boldsymbol{\Sigma}$. The $\boldsymbol{QR}$-Factorization is done once in the same fashion as in item 1.1. - 1.2..

Due to inherent normalization, fast multiplication and storage efficiency quaternions are preferred over actual rotation matrices. In item 3 a $\boldsymbol{QR}$-Factorization is preferred over a column normalization of $\boldsymbol{\Sigma}\boldsymbol{U}$ due to its inaccuracy at near-zero singular values. In general $\boldsymbol{R}$ is an upper triangular matrix. In item 3 it reduces to being diagonal. [Jia+16]

4.3 Constitutive Models

It is common practice in elastoplascity theory to decompose the deformation gradient in its elastic and plastic parts $\boldsymbol{F} = \boldsymbol{F}_E\boldsymbol{F}_P$. The elastic part is recoverable. The plastic part is irreversibly lost. Plastic models are not covered here. They extend the energy densities discussed here with additional terms $\Psi(\boldsymbol{F}_E, \boldsymbol{F}_P)$. For more information refer to the literature [Öch14].

4.3.1 Linear Elasticity

The first aim will be to find the strain energy density $\Psi(\boldsymbol{\epsilon})$ of the strain $\boldsymbol{\epsilon}$. The most general linear stress-strain relationship is given by Hooke's Law in three dimensions:

$$\sigma_{ij} = C_{ijkl}\epsilon_{kl}, \tag{31}$$

where $\boldsymbol{\sigma}$ and $\boldsymbol{\epsilon}$ are second-order tensors with (3x3) = 9 elements. C_{ijkl} is a fourth-order stiffness-tensor with (3x3)x(3x3) = 81 elements. Assuming following symmetries reduces the tensors to 6 and 21 unique elements respectively:

1. Conservation of angular momentum: $\sigma_{ij} = \sigma_{ji} \Rightarrow C_{ijkl} = C_{jikl}$
2. No-net-rotation: $\epsilon_{kl} = \epsilon_{lk} \Rightarrow C_{ijkl} = C_{ijlk}$
3. Equivalence of second-order mixed partials of Ψ:

$$C_{ijkl} = \frac{\partial^2 \Psi}{\partial \epsilon_{kl} \partial \epsilon_{ij}} = \frac{\partial^2 \Psi}{\partial \epsilon_{ij} \partial \epsilon_{kl}} = C_{klij}, \tag{32}$$

where item 3 holds for the strain energy density functional of an (hyper-)elastic material. The stress may then also be calculated by

$$\sigma_{ij} = \sigma_{ij}(\boldsymbol{\epsilon}) = \frac{\partial \Psi}{\partial \epsilon_{ij}}, \tag{33}$$

if such a Ψ is given. An isotropic (direction-independent) linear elastic material further only has three unique elements C_{ijkl}. Using Voigt-Notation, which collapses indices $i = j$ and $k = l$, equation 31 can be rewritten as:

$$\begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{23} \\ \sigma_{13} \\ \sigma_{12} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} & C_{12} & 0 & 0 & 0 \\ & C_{11} & C_{12} & 0 & 0 & 0 \\ & & C_{11} & 0 & 0 & 0 \\ & & & C_{22} & 0 & 0 \\ & & & & C_{22} & 0 \\ sym & & & & & C_{22} \end{bmatrix} \begin{bmatrix} \epsilon_{11} \\ \epsilon_{22} \\ \epsilon_{33} \\ \gamma_{23} \\ \gamma_{13} \\ \gamma_{12} \end{bmatrix}. \tag{34}$$

Experimental results of Hooke's law commonly give

$$\boldsymbol{\epsilon} = \epsilon_{ij} = \frac{1}{E}\left[(1+\nu)\sigma_{ij} - \nu\sigma_{kk}\delta_{ij}\right] = \boldsymbol{C}^{-1}\boldsymbol{\sigma} \tag{35}$$

using engineering constants Young's modulus E and Poisson ratio ν. Inverting $\boldsymbol{C}^{-1}$ and switching to Lamé parameters λ and μ results in the equation:

$$\sigma_{ij} = 2\mu\epsilon_{ij} + \lambda \mathrm{tr}(\boldsymbol{\epsilon})\delta_{ij}. \tag{36}$$

Comparing with equation 34 leads to coefficients $\gamma_{ij} = 2\epsilon_{ij}$ for $(i \neq j)$, $C_{11} = \lambda + 2\mu$, $C_{12} = \lambda$ and $C_{22} = \mu$. γ is also referred to as the engineering strain. Due to the relationship in 33 the model for linear elasticity in terms of the strain energy density function Ψ_{LE} after integration of 36 concludes to:

$$\Psi_{LE} = \mu\|\boldsymbol{\epsilon}\|_F^2 + \frac{\lambda}{2}\mathrm{tr}^2(\boldsymbol{\epsilon}). \tag{37}$$

[AER]

4.3.2 Corotated Hyperelasticity

The simplest tensor assumed by infinitesimal strain theory is the small strain tensor:

$$\boldsymbol{\epsilon} = \frac{1}{2}\left(\boldsymbol{F}_E + \boldsymbol{F}_E^T\right) - \boldsymbol{I} \tag{38}$$

The energy produced by equation 37 using the small strain tensor is not rotationally invariant, w.r.t. to $\boldsymbol{F}_E$: $\Psi_{LE}(\boldsymbol{\epsilon}(\boldsymbol{R}_0\boldsymbol{F}_E)) \neq \Psi_{LE}(\boldsymbol{\epsilon}(\boldsymbol{F}_E))$. Rigid body motions however don't result in strain and consequently don't need to be recovered from. So energy should not change. Given the polar decomposition $\boldsymbol{F}_E = \boldsymbol{R}_E\boldsymbol{S}_E$, an alternate strain measure may be defined as:

$$\boldsymbol{\epsilon}(\boldsymbol{F}_E) = \boldsymbol{\epsilon}\left(\boldsymbol{R}_E^T\boldsymbol{F}_E\right) = \frac{1}{2}\left(\boldsymbol{R}_E^T\boldsymbol{F}_E + \left(\boldsymbol{R}_E^T\boldsymbol{F}_E\right)^T\right) - \boldsymbol{I} = \boldsymbol{S}_E - \boldsymbol{I} \tag{39}$$

Substituting $\boldsymbol{\epsilon}$ into equation 37 leads to the energy definition of corotational hyperelasticity:

$$\Psi_{CH} = \mu\|\boldsymbol{S}_E - \boldsymbol{I}\|_F^2 + \frac{\lambda}{2}\operatorname{tr}^2(\boldsymbol{S}_E - \boldsymbol{I}) \tag{40}$$

Using rotation-invariance of the Frobenius norm:

$$\Psi_{CH} = \mu\|\boldsymbol{F}_E - \boldsymbol{R}_E\|_F^2 + \frac{\lambda}{2}\operatorname{tr}^2\left(\boldsymbol{R}_E^T\boldsymbol{F}_E - \boldsymbol{I}\right) \tag{41}$$

Even more insight yields the relationship to their singular values σ_i, also called the principal stretches:

$$\operatorname{tr}(\boldsymbol{S}) = \sum_{i=1}\sigma_i = \operatorname{tr}(\boldsymbol{\Sigma}_E)$$

$$\|\boldsymbol{S}_E\|_F^2 = \sum_{i=1}\sigma_i^2 = \|\boldsymbol{\Sigma}_E\|_F^2$$

$$\Rightarrow \|\boldsymbol{S}_E - \boldsymbol{I}\|_F^2 = \|\boldsymbol{S}_E\|_F^2 - 2\operatorname{tr}(\boldsymbol{S}_E) + \|\boldsymbol{I}\|_F^2 = \|\boldsymbol{\Sigma}_E - \boldsymbol{I}\|_F^2$$

$$\Psi_{CH}(\boldsymbol{\Sigma}_E) = \mu\|\boldsymbol{\Sigma}_E - \boldsymbol{I}\|_F^2 + \frac{\lambda}{2}\operatorname{tr}^2(\boldsymbol{\Sigma}_E - \boldsymbol{I}) \tag{42}$$

Equation 42 uses the diagonal matrix $\boldsymbol{\Sigma}_E$ containing the principal stretches typically acquired by the singular value decomposition $\boldsymbol{F}_E = \boldsymbol{U}\boldsymbol{\Sigma}_E\boldsymbol{V}^{\boldsymbol{T}}$. Since the energy density in 42 is a function of only three (singular) values, which describe stretch/compression of the material, isotropy of the material is underlined. [McA+11b]

4.3.3 Fixed Corotated Hyperelasticity

Numerical implicit stepping algorithms mostly rely on using the first and second derivative of Ψ, chapter 4.6.6. Furthermore one may describe an isotropic elastic model due to their principal stretches $\sigma_1, \sigma_2, \sigma_3$, chapter 4.2. The rest configuration is reached, when $\sigma_1, \sigma_2, \sigma_3 = 1$. I.e. no elastic forces will be exerted, because the material is completely relaxed (again).

Under material relaxation these stepping algorithms are attracted to a material-dependent primary contour. For instance for a model that aims for high volume preservation (Poisson ratio $\nu = 0.5$) the primary contour is primarily enforcing volume-preservation ($J_E = \sigma_1\sigma_2\sigma_3 \approx 1$). Reaching the rest configuration is only the secondary goal.

The primary contour can be formalized as $\boldsymbol{v} \cdot \boldsymbol{g} = 0$ of the gradient $g_i = \frac{\partial \Psi}{\partial \sigma_i}$ and the eigenvector with the largest-magnitude eigenvalue $\boldsymbol{v}$ of the Hessian $H_{ij} = \frac{\partial \Psi}{\partial \sigma_i \sigma_j}$. Problems arise in Ψ_{CH} as the primary contour can easily cross into the inverted region ($\sigma_3 < 0$) leading to inverted configurations while relaxing. This is undesired behavior. Furthermore the primary contour can lead into energy kinks at extreme stretches leading to oscillations.

Therefore they propose to use the Fixed Corotated Hyperelasticity energy density:

$$\Psi_{FCH} = \mu \|\boldsymbol{\Sigma}_E - \boldsymbol{I}\|_F^2 + \frac{\lambda}{2}(J_E - 1)^2. \tag{43}$$

This leads to the primary contour $J = 1$, which does not cross the inverted region as only one singular value may be negative. [Sto+12]

4.4 Governing equations

deriving the weak form

4.4.1 Conservation of mass

Let the Eulerian mass density be ${}_t\rho({}_t\boldsymbol{x}, t)$ and its (Lagrangian) pull back be ${}_0\rho({}_0\boldsymbol{x}, t)$. After 23 they are related as:

$$\int_{{}_tB_\epsilon} {}_t\rho({}_t\boldsymbol{x}, t) d_t\boldsymbol{x} = \int_{{}_0B_\epsilon} {}^t_0J_0\rho({}_0\boldsymbol{x}, t) d_0\boldsymbol{x} \tag{44}$$

An open ball ${}_0B_\epsilon$ in the reference configuration will have the same mass as its respective open ball B_ϵ^t in the current configuration. Keep in mind that both refer to a deformed state.

Conservation of mass dictates dictates that mass does not depend on motion or time. Only the space occupied by this mass may be more or less.

$$\frac{d}{dt} \int_{{}_tB_\epsilon} {}_t\rho({}_t\boldsymbol{x}, t) d_t\boldsymbol{x} = 0 \tag{45}$$

Equivalently this can be formulated with the constant undeformed initial mass in Lagrangian view:

$$\left(\int_{{}_tB_\epsilon} {}_t\rho({}_t\boldsymbol{x}, t) d_t\boldsymbol{x} \overset{44}{=} \right) \int_{{}_0B_\epsilon} {}^t_0J_0\rho({}_0\boldsymbol{x}, t) d_0\boldsymbol{x} = \int_{{}_0B_\epsilon} {}_0\rho({}_0\boldsymbol{x}, 0) d_0\boldsymbol{x}. \tag{46}$$

In Eulerian view the conservation of mass is more difficult to develop and starts with the Lagrangian view. Since the integrals do account for arbitrary volumes, they are left out in the following:

$$\frac{\partial}{\partial t}({}_0\rho_0^t J) = \frac{\partial_0 \rho}{\partial t}{}_0^t J + \frac{\partial_0^t J}{\partial t}{}_0\rho = 0. \tag{47}$$

The left side could be immediately pushed forward, the right side is harder:

$$\frac{\partial J}{\partial t} = \frac{\partial J}{\partial F_{ij}}\frac{\partial F_{ij}}{\partial t} \overset{49,50}{=} JF_{ji}^{-1}\frac{\partial v_i}{\partial x_k}F_{kj} = J\delta_{ik}\frac{\partial v_i}{\partial x_k} = J\frac{\partial v_i}{\partial x_i}. \tag{48}$$

The determinant differentiation rule can be shown by expressing the determinant with Laplace's expansion and applying the derivative on it:

$$\frac{\partial J}{\partial F_{ij}} = \frac{\partial(F_{ik}\text{adj}(F)_{ki})}{\partial F_{ij}} = \text{adj}(F)_{ji} = JF_{ji}^{-1}. \tag{49}$$

The time-evolution of the deformation gradient is:

$$\frac{\partial_0^t F_{ij}}{\partial t} = \frac{\partial}{\partial t}\frac{\partial_0^t \phi_i}{\partial_0 x_j}({}_0\boldsymbol{x}, t) = \frac{\partial_0 v_i}{\partial_0 x_j}({}_0\boldsymbol{x}, t) = \frac{\partial_t v_i}{\partial_t x_k}({}_0^t\phi({}_0\boldsymbol{x}, t), t)\frac{\partial_t x_k}{\partial_0 x_j}({}_0\boldsymbol{x}, t). \tag{50}$$

Pushing forward 47 with the result of 48 using the material derivative formulation 20:

$$\frac{D}{Dt}\rho({}_t\boldsymbol{x}, t) + \rho({}_t\boldsymbol{x}, t)\vec{\nabla}\cdot\boldsymbol{v}({}_t\boldsymbol{x}, t) = 0 \tag{51}$$

Commonly used is the Nabla operator: $\vec{\nabla} = \left(\frac{\partial}{\partial x_1}, \ldots, \frac{\partial}{\partial x_n}\right)$. [Jia+16]

4.4.2 Conservation of momentum

Continuum forces are divided up into body and surface forces. A surface force acts upon the surface of the material $\partial_t B_\epsilon$. While a body force scales upon the volume of the material ${}_tB_\epsilon$. Conservation of momentum may then be expressed in a similar way to the conservation of mass as:

$$\frac{d}{dt}\int_{{}_tB_\epsilon}\rho({}_t\boldsymbol{x}, t)\boldsymbol{v}({}_t\boldsymbol{x}, t)d_t\boldsymbol{x} = \int_{\partial_t B_\epsilon}\boldsymbol{\sigma}d_t\boldsymbol{A}({}_t\boldsymbol{x}) + \int_{{}_tB_\epsilon}\boldsymbol{f}^{\text{body}}d_t\boldsymbol{x}. \tag{52}$$

Where the assumption due to angular momentum conservation $\boldsymbol{\sigma}^T = \boldsymbol{\sigma}$ out of the second part of this section has already been made for completion. Beginning with a mix out of Lagrangian and Eulerian view

$$\left(\frac{d}{dt}\int_{{}_tB_\epsilon}\rho({}_t\boldsymbol{x}, t)\boldsymbol{v}({}_t\boldsymbol{x}, t)d_t\boldsymbol{x} \overset{19}{=}\right)\frac{d}{dt}\int_{{}_0B_\epsilon}{}_0^t J\rho({}_0\boldsymbol{x}, t)\boldsymbol{v}({}_0\boldsymbol{x}, t)d_0\boldsymbol{x}$$

$$\overset{47}{=}\int_{{}_0B_\epsilon}{}_0^t J\rho({}_0\boldsymbol{x}, t)\boldsymbol{a}({}_0\boldsymbol{x}, t)d_0\boldsymbol{x} = \int_{\partial_t B_\epsilon}\boldsymbol{\sigma}d_t\boldsymbol{A}({}_t\boldsymbol{x}) + \int_{{}_tB_\epsilon}\boldsymbol{f}^{\text{body}}d_t\boldsymbol{x}, \tag{53}$$

where conservation of mass (equation 47) was applied. The Eulerian push-forward of the left side combined with the divergence theorem becomes:

$$\int_{{}_tB_\epsilon} \rho({}_t\boldsymbol{x},t)\boldsymbol{a}({}_t\boldsymbol{x},t)d_t\boldsymbol{x} = \int_{\partial_t B_\epsilon} \vec{\nabla}\cdot\boldsymbol{\sigma}d_t\boldsymbol{x} + \int_{{}_tB_\epsilon} \boldsymbol{f}^{\text{body}}d_t\boldsymbol{x}. \tag{54}$$

The acceleration ${}_t\boldsymbol{a}$ is again defined due to the material derivative 20. Thus the Eulerian momentum balance equation becomes:

$${}_t\rho\frac{D_t\boldsymbol{v}}{Dt} = \vec{\nabla}\cdot\boldsymbol{\sigma} + \boldsymbol{f}^{\text{body}}. \tag{55}$$

There is a quantity left to be defined for the Lagrangian view. The Cauchy stress $\boldsymbol{\sigma}$ is defined in the current configuration. Pulling back the Cauchy stress leads to a stress measure named the first Piola Kirchhoff stress

$$\int_{\partial_t B_\epsilon} {}_t\boldsymbol{\sigma}d_t\boldsymbol{A}({}_t\boldsymbol{x}) \overset{26}{=} \int_{\partial_0 B_\epsilon} {}_0\boldsymbol{\sigma}\boldsymbol{F}^{-T}Jd_0\boldsymbol{A}({}_0\boldsymbol{x}) = \int_{\partial_0 B_\epsilon} \boldsymbol{P}d_0\boldsymbol{A}({}_0\boldsymbol{x}), \tag{56}$$

denoted in the literature as:

$$\boldsymbol{P} = \frac{\partial\Psi}{\partial\boldsymbol{F}} = \boldsymbol{\sigma}\boldsymbol{F}^{-T}J. \tag{57}$$

Summarized the Lagrangian view of the momentum equation with an initial momentum is (${}^0_0J = 1$):

$${}_0\rho({}_0\boldsymbol{x},0){}_0\boldsymbol{a}({}_0\boldsymbol{x},t) = \vec{\nabla}\cdot\boldsymbol{P}({}_0\boldsymbol{x},t) + {}_0\boldsymbol{f}^{\text{body}}({}_0\boldsymbol{x},t){}^t_0J. \tag{58}$$

For the stress strain relationship $\boldsymbol{C}$ in chapter 4.3.1 as well as the start of this chapter conservation of angular momentum caused $\sigma_{ij} = \sigma_{ji}$. This can be shown as follows. The description of angular momentum follows that of linear momentum where product rule generally applies $\left(\frac{D_t\boldsymbol{x}}{Dt} \times {}_t\boldsymbol{v} = {}_t\boldsymbol{v} \times {}_t\boldsymbol{v} = 0\right)$:

$$\int_{{}_tB_\epsilon} {}_t\boldsymbol{x} \times \rho({}_t\boldsymbol{x},t)\boldsymbol{a}({}_t\boldsymbol{x},t)d_t\boldsymbol{x} = \int_{\partial_t B_\epsilon} {}_t\boldsymbol{x} \times \boldsymbol{\sigma}^T d_t\boldsymbol{A}({}_t\boldsymbol{x}) + \int_{{}_tB_\epsilon} {}_t\boldsymbol{x} \times \boldsymbol{f}^{\text{body}}d_t\boldsymbol{x}. \tag{59}$$

In component form this becomes:

$$\int_{{}_tB_\epsilon} \varepsilon_{ijk}\rho x_j a_k d_t\boldsymbol{x} = \int_{\partial_t B_\epsilon} \varepsilon_{ijk}x_j\sigma_{mk}dA_m({}_t\boldsymbol{x}) + \int_{{}_tB_\epsilon} \varepsilon_{ijk}x_j f^{\text{body}}_k d_t\boldsymbol{x}. \tag{60}$$

The divergence theorem is again applied to the surface forces $\left(\vec{\nabla} = \frac{\partial}{\partial x_m}\right)$:

$$\begin{aligned}\int_{\partial_t B_\epsilon} \varepsilon_{ijk}x_j\sigma_{mk}dA_m({}_t\boldsymbol{x}) &= \int_{{}_tB_\epsilon} \varepsilon_{ijk}\frac{\partial(x_j\sigma_{mk})}{\partial x_m}d_t\boldsymbol{x} \\ &= \int_{{}_tB_\epsilon} \varepsilon_{ijk}\left(\delta_{jm}\sigma_{mk} + x_j\frac{\partial\sigma_{mk}}{\partial x_m}\right)d_t\boldsymbol{x}.\end{aligned} \tag{61}$$

The conservation of momentum (eq. 54) can then be applied to the result of plugging 61 back into 60 leaving only:

$$\int_{{}_tB_\epsilon} \varepsilon_{ijk}\sigma_{jk} d_t\boldsymbol{x} = 0. \tag{62}$$

Leaving out the integral and multiplying by ε_{irs} enables equation 12. The Cauchy stress is constrained to

$$\sigma_{ij} = \sigma_{ji} \tag{63}$$

as assumed before. Note that $\boldsymbol{P}$ however is not constrained to be symmetric.

4.4.3 Weak formulation

Before deriving the weak form an explanation of what it achieves is in order.

The previous presented governing equations are written in the strong form: a solution to the equation needs to be exact on the whole domain (and is as such influenced by the whole domain in general). Such a solution can be found for simplified models analytically. These act as a ground truth to numerical methods like the material point method. Analytical methods of today can't handle complex problems: Numerical solutions try to overcome that hurdle.

The complete mathematical description of the weak formulation is beyond this thesis. Numeric books that discuss finite element methods will provide one ([Bat06], [DR08]). For simplicity: The weak formulation combined with Galerkin discretization restricts the globality of the strong method due to so called 'test functions': ${}_0\boldsymbol{q}_h$. A 'test function' generally only has limited support, i.e. ${}_0\boldsymbol{q}_h \neq 0$ on a very small subset of the whole domain Ω^0. This is mostly used to gather information on a local neighborhood.

The weak formulation requires an observation of the dot product. Consider the conservation of momentum in the strong form as formulated before (58) as

$${}_0\rho({}_0\boldsymbol{x}, 0){}_0\boldsymbol{a}({}_0\boldsymbol{x}, t) = \vec{\nabla} \cdot \boldsymbol{P}({}_0\boldsymbol{x}, t) + {}_0\boldsymbol{f}^{\text{body}}({}_0\boldsymbol{x}, t){}_0^tJ. \tag{64}$$

Multiply both sides with the dot product of an arbitrary function ${}_0\boldsymbol{q}(\cdot, t) : \Omega^0 \to \mathbb{R}^d$ and integrate over Ω^0. If a solution solves the balance of 64 then it also solves:

$$\int_{\Omega^0} {}_0q_i({}_0\boldsymbol{x}, t) \left({}_0\rho({}_0\boldsymbol{x}, 0){}_0a_i({}_0\boldsymbol{x}, t) - {}_0\boldsymbol{f}^{\text{body}}({}_0\boldsymbol{x}, t){}_0^tJ \right) d_0\boldsymbol{x}$$

$$= \int_{\Omega^0} {}_0q_i({}_0\boldsymbol{x}, t) \frac{\partial P_{ij}}{\partial x_j}({}_0\boldsymbol{x}, t) d_0\boldsymbol{x}. \tag{65}$$

With the help of partial integration in multiple dimension the derivative moves over to the test-function. Balancing out the order of derivatives, by moving a derivative to the test function, is the main motive of the weak form:

$$\int_{\Omega^0} \frac{\partial({}_0q_i({}_0\boldsymbol{x}, t) P_{ij}({}_0\boldsymbol{x}, t))}{\partial_0 x_j} - \frac{\partial_0 q_i}{\partial_0 x_j}({}_0\boldsymbol{x}, t) P_{ij}({}_0\boldsymbol{x}, t) d_0\boldsymbol{x}$$

The divergence theorem allows to convert the first term to a boundary integral.

$$\int_{\partial\Omega^0} {}_0q_i({}_0\boldsymbol{x},t)P_{ij}({}_0\boldsymbol{x},t)d_0\boldsymbol{A}({}_0\boldsymbol{x}) - \int_{\Omega^0} \frac{\partial_0 q_i}{\partial_0 x_j}({}_0\boldsymbol{x},t)P_{ij}({}_0\boldsymbol{x},t)d_0\boldsymbol{x} \tag{66}$$

Mathematically the boundary integral serves as a boundary condition which is set by the specific problem (e.g. context of the simulation). Finally putting together the previous results gives the weak form of force balance in the Lagrangian view:

$$\int_{\Omega^0} {}_0q_i({}_0\boldsymbol{x},t)\left({}_0\rho({}_0\boldsymbol{x},0){}_0a_i({}_0\boldsymbol{x},t) - {}_0f_i^{\text{body}}({}_0\boldsymbol{x},t){}_0^tJ\right)d_0\boldsymbol{x}$$

$$= \int_{\partial\Omega^0} {}_0q_i({}_0\boldsymbol{x},t)P_{ij}({}_0\boldsymbol{x},t)d_0\boldsymbol{A}({}_0\boldsymbol{x}) - \int_{\Omega^0} \frac{\partial_0 q_i}{\partial_0 x_j}({}_0\boldsymbol{x},t)P_{ij}({}_0\boldsymbol{x},t)d_0\boldsymbol{x}. \tag{67}$$

In the material point method stress computations are more naturally done in the current configuration, or equally in terms of the Cauchy stress as seen in chapter 4.3.1. Pushing the equation forward to Eulerian view with the push forward ${}_t\boldsymbol{q} : \Omega^t \to \mathbb{R}^d$ of ${}_0\boldsymbol{q}$ is only a problem for the last term:

$$\int_{\Omega^0} \frac{\partial_0 q_i}{\partial_0 x_j}({}_0\boldsymbol{x},t)P_{ij}d_0\boldsymbol{x} \overset{57,23}{=} \int_{\Omega^0} \left({}_0^tF_{kj}\frac{\partial_t q_i}{\partial_t x_k}({}_t\boldsymbol{x},t)\right)\left({}_t^0F_{kj}\sigma_{ik}{}_0^tJ\right){}_t^0Jd_t\boldsymbol{x}$$

$$= \int_{\Omega^t} \frac{\partial_t q_i}{\partial_t x_k}({}_t\boldsymbol{x},t)\sigma_{ik}d_t\boldsymbol{x}. \tag{68}$$

This completes the Eulerian view to be:

$$\int_{\Omega^t} {}_tq_i({}_t\boldsymbol{x},t)\left({}_t\rho({}_t\boldsymbol{x},t){}_ta_i({}_t\boldsymbol{x},t) - {}_tf_i^{\text{body}}({}_t\boldsymbol{x},t)\right)d_t\boldsymbol{x}$$

$$= \int_{\partial\Omega^t} {}_tq_i({}_t\boldsymbol{x},t)\sigma_{ij}d_tA_j({}_t\boldsymbol{x}) - \int_{\Omega^t} \frac{\partial_t q_i}{\partial_t x_k}({}_t\boldsymbol{x},t)\sigma_{ik}d_t\boldsymbol{x}. \tag{69}$$

[Jia+16] [Str07] [Bat06]

4.5 Material Point Method

The key idea of the material point method is to use (Lagrangian) particles as a consistent storage of material properties. All stress based forces are computed on a Eulerian grid however. This grid does not store any material properties and is therefore often referred to as a scratch pad.

As a corollary there needs to be a way to transfer information from a particle to the neighboring grid cells. This also induces switching from Lagrangian to Eulerian view. After stresses are computed there also needs to be a way to get back the relevant information from the grid to the particles. After the information is transferred back the particles are moved. This advection is typically hard to do in Eulerian/FEM-like methods and cause of a lot of problems down the development pipeline. In a Lagrangian view particle advection is trivial.

Its is very important that the two transfers as well as the grid solver are in compliance with all governing equations. While the grid solver will be derived from the weak form of the governing equation the transfers also need to be chosen in a way that also conserve the properties defined in them. [Jia+16]

4.5.1 Interpolation weights

The choice of interpolation weights is flexible. Nevertheless a kernel $w(\boldsymbol{x})$ requires some important properties to be qualified for MPM:

1. Partition of unity:
$$\sum_i w\left(\boldsymbol{x} - \boldsymbol{x}_i^n\right) = 1. \tag{70}$$

2. Identity relation:
$$\sum_i \boldsymbol{x}_i^n w\left(\boldsymbol{x} - \boldsymbol{x}_i^n\right) = \boldsymbol{x}. \tag{71}$$

3. Non-negativity: $w \geq 0$. Negative weights can cause severe instability or non-physical behavior. [Gao+17]

4. Limited local support to reduce the number of discretizations, chapter 4.6.

5. C^1-continuity s.t. ∇w is continuous.

As a reminder of chapter 3: $\boldsymbol{x}_p$ refers to a particle's position. $\boldsymbol{x}_i$ to a grid cell's position. For a more general discussion refer to [Gao+17]. Often dyadic products of one-dimensional interpolation functions suffice

$$w(\boldsymbol{x} - \boldsymbol{x}_i^n) = w_i^n(\boldsymbol{x}) = w\left(\frac{1}{h}\left(x - x_i^n\right)\right) w\left(\frac{1}{h}\left(y - y_i^n\right)\right) w\left(\frac{1}{h}\left(z - z_i^n\right)\right) \tag{72}$$

$$\nabla w_i^n(\boldsymbol{x}) = \frac{1}{h}\begin{pmatrix} w'(\frac{1}{h}(x - x_i^n))w(\frac{1}{h}(y - y_i^n))w(\frac{1}{h}(z - z_i^n) \\ w(\frac{1}{h}(x - x_i^n))w'(\frac{1}{h}(y - y_i^n))w(\frac{1}{h}(z - z_i^n) \\ w(\frac{1}{h}(x - x_i^n))w(\frac{1}{h}(y - y_i^n))w'(\frac{1}{h}(z - z_i^n) \end{pmatrix}, \tag{73}$$

where h is the grid spacing (of a uniform grid). An interpolation function often employed is a cubic B-spline (C^2):

$$w(x) = \begin{cases} \frac{1}{2}|x|^3 - |x|^2 + \frac{2}{3} & 0 \leqslant |x| < 1 \\ \frac{1}{6}(2 - |x|)^3 & 1 \leqslant |x| < 2 \\ 0 & 2 \leqslant |x| \end{cases}. \tag{74}$$

Since these function are used to weight (or filter) particles a shortening in notation may be employed as $w_{ip}^n = w(\boldsymbol{x}_p^n - \boldsymbol{x}_i^n)$. [JST17][Jia+16][SKB08][Gao+17]

4.5.2 Mass transfer

Each material point will be assigned an initial volume ${}_0V_p$ as well as a initial mass ${}_0m_p$. The volume the material point occupies may change in time due to eq. 24: ${}^t_0J_0V_p = {}_tV_p$. But due to conservation of mass 46 it will have a constant, initial mass associated with it. I.e. there will only be a transfer to the grid and no transfer back. A transfer of mass to the grid may then be expressed as:

$$m_i = \sum_p w_{ip} m_p. \tag{75}$$

$\sum_i m_i = \sum_p m_p$ is a complete prove this fulfills the conservation of mass. Remember there is no transfer back. In doing so no information can be lost on the particles. Proving that the mass transfer to the grid is conserving is enough. The stress based-solver may manipulate this information further (mass-lumping strategies).

$$\sum_i m_i \overset{75}{=} \sum_i \sum_p w_{ip} m_p = \sum_p m_p \sum_i w_{ip} \overset{70}{=} \sum_p m_p \tag{76}$$

[JST17][Jia+16]

4.5.3 APIC transfers

The momentum transfer round trip could be defined as:

1. Particle to grid momentum transfer:

$$(m\boldsymbol{v})_i^n = \sum_p w_{ip} m_p \boldsymbol{v}_p^n. \tag{77}$$

2. Factoring out mass:

$$\boldsymbol{v}_i^n = \frac{(m\boldsymbol{v})_i^n}{m_i^n}. \tag{78}$$

3. Coupled with either ($\alpha \in \{0, 1\}$) or a combination ($\alpha \in]0; 1[$) of:

$$\boldsymbol{v}_p^{n+1} = \alpha \boldsymbol{v}_{p,PIC}^{n+1} + (1-\alpha)\boldsymbol{v}_{p,FLIP}^{n+1} \tag{79}$$

$$\boldsymbol{v}_{p,PIC}^{n+1} = \sum_i w_{ip} \boldsymbol{v}_i^{n+1}. \tag{80}$$

$$\boldsymbol{v}_{p,FLIP}^{n+1} = \boldsymbol{v}_p^n + \sum_i w_{ip}(\boldsymbol{v}_i^{n+1} - \boldsymbol{v}_i^{n+1}). \tag{81}$$

While PIC-Transfers are very stable they suffer from excessive (energy) dissipation due to double interpolating on the whole quantity 77,80. This causes a heavy loss in angular momentum and velocity modes. $FLIP$-Transfers avoid dissipation and loss of angular momentum by only updating the velocity with a difference

81. However some velocity modes are also not recognized on the grid and may cause unpredictable and unstable behavior in following steps. Therefore often a combination of both is taken.

$APIC$ builds on top of the very stable PIC-transfers and effectively only adds an extra term of the Taylor series to increase accuracy. This extra term $\boldsymbol{C}_p$ may be in short just referred to as the velocity derivative. The local velocity field around a particle may then be characterized by the affine function $\boldsymbol{v}(\boldsymbol{x}) = \boldsymbol{C}_p(\boldsymbol{x} - \boldsymbol{x}_p)$.

Motivated by the theory of angular momentum and moment of inertia one can define a quantity

$$\boldsymbol{D}_p^n = \sum_i w_{ip}^n (\boldsymbol{x}_i^n - \boldsymbol{x}_p^n)(\boldsymbol{x}_i^n - \boldsymbol{x}_p^n)^T, \tag{82}$$

which is similar to the classically known inertia tensor:

$$\boldsymbol{I}_p = -\sum_i m_i [\boldsymbol{x}_i - \boldsymbol{x}_p][\boldsymbol{x}_i - \boldsymbol{x}_p]^T \tag{83}$$

$$= \sum_i m_i ((\boldsymbol{x}_i - \boldsymbol{x}_p)^T (\boldsymbol{x}_i - \boldsymbol{x}_p)\boldsymbol{I} - (\boldsymbol{x}_i - \boldsymbol{x}_p)(\boldsymbol{x}_i - \boldsymbol{x}_p)^T)$$

Bearing in mind that $\boldsymbol{D}_p^n$ does not include a mass and is defined for an affine motion instead of an angular motion, where $[a]_{\alpha\gamma} = \varepsilon_{\alpha\beta\gamma} a_\beta$ is the cross-product matrix and $\boldsymbol{I}$ denotes the identity matrix. In classical mechanics the angular velocity $\boldsymbol{\omega_p}$ can be then described using the inertia tensor $\boldsymbol{I}_p$ with the help of the angular momentum $\boldsymbol{L}_p$:

$$\boldsymbol{\omega}_p = \boldsymbol{I}_p^{-1} \boldsymbol{L}_p. \tag{84}$$

This motivates the velocity derivative to be equally defined by a similar relationship, where $\boldsymbol{B}_p^n$ holds momentum information.

$$\boldsymbol{C}_p^n = (\boldsymbol{D}_p^n)^{-1} \boldsymbol{B}_p^n. \tag{85}$$

The transfers of the $APIC$-scheme are then summarized:

1. Particle to grid:

$$(m\boldsymbol{v})_i^n = \sum_p w_{ip}^n m_p (\boldsymbol{v}_p^n + \boldsymbol{B}_p^n (\boldsymbol{D}_p^n)^{-1} (\boldsymbol{x}_i^n - \boldsymbol{x}_p^n)) \tag{86}$$

2. Factoring out mass:

$$\boldsymbol{v}_i^n = \frac{(m\boldsymbol{v})_i^n}{m_i^n}. \tag{87}$$

3. Grid to particle transfer (in a PIC-manner), where in contrast the new particle position $\boldsymbol{x}_p$ also needs to be interpolated:

$$\boldsymbol{x}_p^{n+1} = \sum_i w_{ip} \boldsymbol{x}_i^{n+1} \tag{88}$$

$$\boldsymbol{v}_p^{n+1} = \sum_i w_{ip} \boldsymbol{v}_i^{n+1} \tag{89}$$

$${}^{-}\Delta\boldsymbol{x} = \boldsymbol{x}_i^n - \boldsymbol{x}_p^n + \boldsymbol{x}_i^{n+1} - \boldsymbol{x}_p^{n+1}, \quad {}^{+}\Delta\boldsymbol{x} = \boldsymbol{x}_i^n - \boldsymbol{x}_p^n - \boldsymbol{x}_i^{n+1} + \boldsymbol{x}_p^{n+1}$$

$$\boldsymbol{B}_p^{n+1} = \frac{1}{2} \sum_i w_{ip}^n (\boldsymbol{v}_i^{n+1} ({}^{-}\Delta\boldsymbol{x})^T) + {}^{+}\Delta\boldsymbol{x} (\boldsymbol{v}_i^{n+1})^T). \tag{90}$$

For a full proof that these transfers preserve linear and angular momentum consult [JST17]. For the choice of dyadic products of cubic b-splines (74) $\boldsymbol{D}_p$ takes on the simple form:

$$\boldsymbol{D}_p^n = \frac{1}{3} h^2 \boldsymbol{I}. \tag{91}$$

A simple proof (Appendices: A) cancelling out the numerous polynomials can be done for instance using SymPy ([Sym]).[JST17][Jia+15]

4.5.4 CFL condition

The CFL condition is prominent for FEM-like methods: For a stable integration a particle should not travel father than the grid spacing h in a discrete time-step Δt. Δt is thus limited by:

$$\Delta t \leq \frac{h}{\|\boldsymbol{v}_i^n\|_2}. \tag{92}$$

Assuming $\|\boldsymbol{x}_i^n - \boldsymbol{x}_p^n\| \leq \kappa h$, where κ is determined by the interpolation stencil support (cubic 3D: $\kappa = 2\sqrt{3}$) and additionally assuming $\boldsymbol{D}_p^n = k\boldsymbol{I} \Rightarrow (\boldsymbol{D}_p^n)^{-1} = \frac{1}{k}\boldsymbol{I}$ (cubic 3D: $k = \frac{1}{3}h^2$). $\|\boldsymbol{v}_i^n\|_2$ can be estimated on the particles. Typically the number of particles is lower. Given Eq. 86:

$$\begin{aligned} \|\boldsymbol{v}_i^n\|_2 &\leq \frac{1}{m_i^n} \left(\sum_p w_{ip}^n m_p \|\boldsymbol{v}_p^n\|_2 + \sum_p w_{ip}^n m_p \|\boldsymbol{B}_p^n\|_F \|(\boldsymbol{D}_p^n)^{-1} (\boldsymbol{x}_i^n - \boldsymbol{x}_p^n)\|_2 \right) \\ &\leq \max_p (\|\boldsymbol{v}_p^n\|_2 + \frac{\kappa}{k} \Delta x \|\boldsymbol{B}_p^n\|_F) \end{aligned} \tag{93}$$

[JST17]

4.6 Discretization

The weak form of the force-balance in (67,69) implies the following for MPM preferable description:

$$\int_{\Omega^0} ({}_0q_\alpha)({}_0\rho_0)({}_0a_\alpha) d_0\boldsymbol{x} = \int_{\partial\Omega^{t^n}} {}_tq_\alpha \sigma_{\alpha\beta} d_t A_\beta({}_t\boldsymbol{x}) - \int_{\Omega^{t^n}} \frac{\partial_t q_\alpha}{\partial_t x_\beta} \sigma_{\alpha\beta} d_t\boldsymbol{x}. \tag{94}$$

The boundary integral is mostly due to collisions and will be ignored for now. [Gas+15] discusses level set collisions due to constraint collisions, object penalty collision and penalty self-collisions. An handling of collision would need to take

them into account when solving the equation. A simple though less accurate method is to process collisions separately in a typical computer graphics manner, which is assumed for now.

4.6.1 Discretize time

Any integrator conserving linear and angular momentum could be used to discretize time. The class of time integrators used here are characterized by

$$\frac{y^{n+1}-y^n}{\Delta t} = f^{n+\lambda} = f\left(t^n + \lambda\Delta t, (1-\lambda)y^n + \lambda y^{n+1}\right) \tag{95}$$

for a differential equation of order one:

$$\frac{\partial y}{\partial t}(t) = f(t, y(t)), \quad y(0) = y_0. \tag{96}$$

A prominent member of this class is the implicit midpoint rule ($\lambda = \frac{1}{2}$). Replacing the Lagrangian acceleration ${}_0a_\alpha$ in eq. 94 with the left side of 95 using velocity, taking care of the right side and pushing forward to Eulerian view:

$$\frac{1}{\Delta t}\int_{\Omega^{t^n}} ({}_tq_\alpha)({}_t\rho)({}_tv_\alpha^{n+1} - {}_tv_\alpha^n)d_t\boldsymbol{x} = -\int_{\Omega^{t^n}} \frac{\partial_t q_\alpha}{\partial_t x_\beta}\sigma_{\alpha\beta}^{n+\lambda} d_t\boldsymbol{x}. \tag{97}$$

4.6.2 Discretize space

A Galerkin discretization brings all spatial terms of equation 97 to a finite-dimensional space: $\boldsymbol{q} \to \boldsymbol{q}_h$. To not further clutter up the notation, the h will be omitted. This will replace $q_\alpha, v_\alpha^n, v_\alpha^{n+1}$ with their finite-dimensional grid-based interpolants:

$${}_tq_\alpha^n = ({}_tq_{i\alpha}^n)({}_tw_i), \quad {}_tv_\alpha^n = ({}_tv_{j\alpha}^n)({}_tw_j), \quad {}_tv_\alpha^{n+1} = ({}_tv_{j\alpha}^{n+1})({}_tw_j). \tag{98}$$

Further chapter 4.4.3 mentions that $\boldsymbol{q}$ can be chosen arbitrarily. The Galerkin discretization of a d-dimensional space with m grid nodes therefore uses the standard basis functions $e_1, e_2, ..., e_{d\times m}$. Due to the scalar-product $d \times m$ equations would need to be solved:

$$\frac{1}{\Delta t}\int_{\Omega^{t^n}} ({}_tw_i)({}_t\rho)({}_tw_j)({}_tv_{j\alpha}^{n+1} - {}_tv_{j\alpha}^n)d_t\boldsymbol{x} = -\int_{\Omega^{t^n}} \frac{\partial_t w_i}{\partial_t x_\beta}\sigma_{\alpha\beta} d\boldsymbol{x}. \tag{99}$$

A mass matrix can be factored out as:

$$m_{ij}^n = \int_{\Omega^{t^n}} {}_tw_i({}_t\rho){}_tw_j d_t\boldsymbol{x}. \tag{100}$$

The Lagrangian pull-back relates this to the initial density in the Lagrangian view and discretizing the integral with the initial time-invariant particle mass $m_p \approx V_p^0\rho({}_0x_p, 0)$:

$$m_{ij}^n = \int_{\Omega^{t^0}} ({}_tw_i)({}_0\rho_0)({}_tw_j)d_0\boldsymbol{x} \approx \sum_p m_p w_i(\boldsymbol{x}_p)w_j(\boldsymbol{x}_p). \tag{101}$$

This matrix is symmetric positive semi-definite (since mass is positive). Numerically this matrix is mostly not used as is due to possibility of it being singular. This is solved commonly due to a mass lumping strategy. Replace m_{ii}^n with the i-th row sum to get:

$$\sum_p m_p w_{ip} w_{jp} \overset{70}{\approx} \sum_p m_p w_{ip} \overset{75}{=} m_i^n, \tag{102}$$

where partition of unity $\sum_j w_{jp} = 1$ is used. This is exactly the mass transfer as in eq. 75, s.t. no further assembling of a mass matrix is needed. Discretizing the right side of 99 with an estimated per particle stress $\boldsymbol{\sigma}_p^{n+\lambda}$:

$$\int_{\Omega^{t^n}} \frac{\partial_t w_i}{\partial_t x_\beta} \sigma_{\alpha\beta} d\boldsymbol{x} \approx \sum_p (\sigma_p^{n+\lambda})_{\alpha\beta} \frac{\partial w_{ip}^n}{\partial x_\beta} V_p^n. \tag{103}$$

Setting in equation 102 and 103 into 99 summarizes the space discretization as:

$$\frac{1}{\Delta t}((m^n \boldsymbol{v}^{n+1})_i - (m^n \boldsymbol{v}^n)_i) = -\sum_p \boldsymbol{\sigma}_p^{n+\lambda} \nabla w_{ip}^n V_p^n = \boldsymbol{f}_i^{n+\lambda}. \tag{104}$$

The momentum change of the left side is by construction equal to a (grid node) force.
Given that the material point method keeps track of the deformation by a deformation gradient. Each particle will have one associated with it for the deformation of its local neighborhood $\boldsymbol{F}_p^n$. Based on this one may also gain a volume change measure around the particle as $J_p^n = \det(\boldsymbol{F}_p^n)$. Starting with an initial volume of a particle V_p^0 the volume may be tracked in time by:

$$V_p^n \overset{24}{\approx} V_p^0 J_p^n. \tag{105}$$

In eq. 57 an alternate measure for the stress by the first Piola Kirchhoff stress. The results of 104 may therefore equally expressed by it:

$$\begin{aligned} \boldsymbol{f}_i^{n+\lambda} &\overset{57,105}{=} -\sum_p \frac{1}{J_p^n} \boldsymbol{P}_p^{n+\lambda} (\boldsymbol{F}_p^n)^T \nabla w_{ip} V_p^0 J_p^n \\ &= -\sum_p \boldsymbol{P}_p^{n+\lambda} (\boldsymbol{F}_p^n)^T \nabla w_{ip} V_p^0. \end{aligned} \tag{106}$$

[Jia+16] [JST17] [Bat06]

4.6.3 Deformation gradient evolution

In eq. 50 the evolution of the deformation gradient is shown to be:

$$\frac{\partial_0^t \boldsymbol{F}}{\partial t} = \nabla_0 \boldsymbol{v}({}_0\boldsymbol{x}, t). \tag{107}$$

Discretizing the Lagrangian deformation gradient in time with eq. 95 results in:

$$\frac{\boldsymbol{F}_p^{n+1} + \boldsymbol{F}_p^n}{\Delta t} = \nabla_0 \boldsymbol{v}^{n+\lambda}({}_0\boldsymbol{x}). \tag{108}$$

Pushing the right side forward to Eulerian view

$$\frac{\boldsymbol{F}_p^{n+1} + \boldsymbol{F}_p^n}{\Delta t} = \nabla_t \boldsymbol{v}^{n+\lambda}({}_t\boldsymbol{x}){}_0^t\boldsymbol{F} = \nabla_t \boldsymbol{v}^{n+\lambda}({}_t\boldsymbol{x})\boldsymbol{F}_p^n \tag{109}$$

and further applying the Galerkin discretization

$$({}_t v^{n+\lambda})_\alpha = (v_i^{n+\lambda})_\alpha w_i \Rightarrow \frac{\partial({}_t v^{n+\lambda})_\alpha}{\partial x_\beta} = (v_i^{n+\lambda})_\alpha \frac{\partial w_i}{\partial x_\beta} \tag{110}$$

leads to the final update rule for the deformation gradient:

$$\boldsymbol{F}_p^{n+1} = \left(\boldsymbol{I} + \Delta t \sum_i \boldsymbol{v}_i^{n+\lambda} (\nabla w_{ip})^T\right) \boldsymbol{F}_p^n. \tag{111}$$

The discretization of the position will also be of need to advance the particles and weight them back:

$$\frac{\partial_t \boldsymbol{x}}{\partial t} = {}_t\boldsymbol{v} \Rightarrow \frac{\hat{\boldsymbol{x}}_i^{n+1} - \boldsymbol{x}_i^n}{\Delta t} = \boldsymbol{v}_i^{n+\lambda}. \tag{112}$$

The grid position $\hat{\boldsymbol{x}}_i^{n+1}$ does not correspond to an actual deformation. The grid never actually gets deformed (unlike in FEM-methods). Therefore the discretized evolution of the deformation gradient 111 is directly a function of $\hat{\boldsymbol{x}}$. For the point $\hat{\boldsymbol{x}}_i^{n+1}$ this becomes:

$$\hat{\boldsymbol{F}}_p^{n+1}(\hat{\boldsymbol{x}}_i^{n+1}) = \left(\boldsymbol{I} + \sum_i (\hat{\boldsymbol{x}}_i^{n+1} - \boldsymbol{x}_i^n)(\nabla w_{ip})^T\right) \boldsymbol{F}_p^n. \tag{113}$$

As part of the class of time integrators in use (95) a function of $\boldsymbol{x}_i$ gets evaluated at an in-between point given by:

$$\boldsymbol{x}_i^{n+\lambda} = \lambda \boldsymbol{x}_i^{n+1} + (1-\lambda)\boldsymbol{x}_i^n. \tag{114}$$

Plugging this point into 113 leads to the following generalization:

$$\begin{aligned}\hat{\boldsymbol{F}}_p^{n+\lambda}(\boldsymbol{x}_i^{n+\lambda}) &= \left(\boldsymbol{I} + \lambda \sum_i (\hat{\boldsymbol{x}}_i^{n+1} - \boldsymbol{x}_i^n)(\nabla w_{ip})^T\right) \boldsymbol{F}_p^n \\ &= (1-\lambda)\boldsymbol{F}_p^n + \lambda \boldsymbol{F}_p^{n+1}\end{aligned} \tag{115}$$

This [Jia+16]

4.6.4 Grid nodal forces

The notion of a total elastic potential energy function Ψ was introduced in chapter 4.3.2. The MPM approximation to this function can be defined by:

$$e(\hat{\boldsymbol{x}}) = \sum_p V_p^0 \Psi(\hat{\boldsymbol{F}}_{Ep}(\hat{\boldsymbol{x}})). \tag{116}$$

The evolution of the deformation gradient 113 for a general $\hat{\boldsymbol{x}}_i$ is necessary

$$\frac{\partial \hat{F}_{\omega\beta}}{\partial \hat{x}_\alpha} = \delta_{\omega\alpha} \frac{\partial w_{ip}}{\partial x_\gamma} F_{\gamma\beta} = \delta_{\omega\alpha} F_{\gamma\beta} \frac{\partial w_{ip}}{\partial x_\gamma} \tag{117}$$

for the spatial derivative of the potential $e(\hat{\boldsymbol{x}})$. This is just the force created by elastic stresses out of eq. 106:

$$\begin{aligned} \frac{\partial e}{\partial \hat{x}_{i\alpha}}(\hat{\boldsymbol{x}}) &= \sum_p V_p^0 \frac{\partial \Psi}{\partial \hat{F}_{\omega\beta}}(\hat{\boldsymbol{F}}_{Ep}) \frac{\partial \hat{F}_{\omega\beta}}{\partial \hat{x}_\alpha} \\ &= \sum_p V_p^0 P_{\alpha\beta}(\hat{\boldsymbol{F}}_{Ep}) F_{\gamma\beta} \frac{\partial w_{ip}}{\partial x_\gamma} = -\hat{f}_{i\alpha}. \end{aligned} \tag{118}$$

The nodal force can be also described in terms of the Cauchy stress:

$$\hat{\boldsymbol{f}}_i = -\sum_p V_p^n \boldsymbol{\sigma}_p(\hat{\boldsymbol{F}}_{Ep}) \nabla w_{ip}^n. \tag{119}$$

Due to eq. 115 the stress computation is summarized for the class of functions in use as:

$$\boldsymbol{P}^{n+\lambda}(\hat{\boldsymbol{F}}_{Ep}) = \boldsymbol{P}(\hat{\boldsymbol{F}}_{Ep}^{n+\lambda}), \quad \boldsymbol{\sigma}^{n+\lambda}(\hat{\boldsymbol{F}}_{Ep}) = \boldsymbol{\sigma}(\hat{\boldsymbol{F}}_{Ep}^{n+\lambda}). \tag{120}$$

[Jia+16][JST17]

4.6.5 Symplectic midpoint scheme

Starting from the last update on the grid which is typically the position with eq. 95 the general midpoint scheme ($\lambda = \frac{1}{2}$) is:

$$\hat{\boldsymbol{x}}_i^{n+1} = \boldsymbol{x}_i^n + \Delta t \boldsymbol{v}_i^{n+\frac{1}{2}}. \tag{121}$$

Due to 104 a velocity update can be put together as:

$$\hat{\boldsymbol{v}}_i^{n+1} = \boldsymbol{v}_i^n + \frac{\Delta t}{m_i^n} \boldsymbol{f}_i^{n+\frac{1}{2}}. \tag{122}$$

A variable of $n + \frac{1}{2}$ then gets evaluated at the midpoint (114):

$$\boldsymbol{f}_i^{n+\frac{1}{2}} = \boldsymbol{f}_i \left(\frac{\boldsymbol{x}_i^n + \hat{\boldsymbol{x}}_i^{n+1}}{2} \right). \tag{123}$$

The modified energy conserving implicit midpoint scheme from [Gon00] for the Material Point Method differs by the use of a trapezoidal approximation of $\boldsymbol{v}_i^{n+\frac{1}{2}}$:

$$\boldsymbol{v}_i^{n+\frac{1}{2}} \approx \frac{\hat{\boldsymbol{v}}_i^{n+1} + \boldsymbol{v}_i^n}{2}. \tag{124}$$

The trapezoidal rule has the same order of error $O(\Delta t^2)$ as the implicit midpoint scheme. This modification allows for a more direct one-step scheme as shown in the following. Plugging $\boldsymbol{x}_i^{n+1}$ of eq. 121 into 122 leads to the discretized momentum equation:

$$\boldsymbol{h}(\hat{\boldsymbol{v}}_i^{n+1}) = m_i^n \frac{\hat{\boldsymbol{v}}_{\boldsymbol{i}}^{n+1} - \boldsymbol{v}_i^n}{\Delta t} - \boldsymbol{f}_i \left(\boldsymbol{x}_i^n + \frac{\Delta t}{4}(\hat{\boldsymbol{v}}_i^{n+1} + \boldsymbol{v}_i^n) \right) = 0. \tag{125}$$

One can recast this method back to an energy function by integrating:

$$E(\boldsymbol{v}_i) = \sum_i \frac{m_i^n}{8} \|\boldsymbol{v}_i - \boldsymbol{v}_i^n\|_2^2 + e(\boldsymbol{x}_i^n + \frac{\Delta t}{4}(\boldsymbol{v}_i + \boldsymbol{v}_i^n)), \tag{126}$$

where e is just the discretized elastic potential energy out of 116. In general this is a optimization objective that needs minimizing to solve for the updated velocities. Recasting allows solving in a general scheme by an optimization integrator.

In general starting off with an energy description is a more physical approach of the problem. The first term can be identified clearly as the kinetic energy. Moreover it easily allows adding potential terms for gravity or collision [Gas+15]. Or modifying the energy function altogether based off for instance the fundamental state of matter: as an example the phase transition to and physics of a liquid [Sto+14]. The analysis of minimizing the objective is equivalent to finding the zero crossing of the derivative:

$$\underset{\forall \boldsymbol{v}_i}{\operatorname{argmin}}(E(\boldsymbol{v}_i)) \Leftrightarrow g(\boldsymbol{v}_i) = \frac{\partial E}{\partial \boldsymbol{v}_i} = 0. \tag{127}$$

Since the objective is minimizing the scale of $\boldsymbol{E}$ can be chosen arbitrarily, i.e. a zero crossing does not scale. The scaling here can be identified as:

$$\boldsymbol{g}(\boldsymbol{v}_i) = \frac{\Delta t}{4} h(\boldsymbol{v}_i) = m_i^n \frac{\boldsymbol{v}_{\boldsymbol{i}} - \boldsymbol{v}_i^n}{4} - \frac{\Delta t}{4} \boldsymbol{f}_i \left(\boldsymbol{x}_i^n + \frac{\Delta t}{4}(\boldsymbol{v}_i + \boldsymbol{v}_i^n) \right). \tag{128}$$

[Jia+16][JST17]

4.6.6 Newton's Method

Many minimization and root finding algorithms are available. One of such is Newton's Method, which allows for rapid quadratic convergence in a near local neighborhood.

$$\boldsymbol{v}_i^{(i+1)} = \boldsymbol{v}_i^{(i)} + \left[\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{v}} \left(\boldsymbol{v}_i^{(i)} \right) \right]^{-1} \boldsymbol{g} \left(\boldsymbol{v}_i^{(i)} \right) \tag{129}$$

Algorithm 1 Conjugate gradient

procedure CONJUGATE-GRADIENT(H, x, f)
  $\boldsymbol{x} \leftarrow$ INITIALGUESS()
  $\boldsymbol{Hx} \leftarrow$ COMPUTEHP(x)
  $\boldsymbol{r} \leftarrow \boldsymbol{f} - \boldsymbol{Hx}$
  $\boldsymbol{p} \leftarrow \boldsymbol{r}$
  $\gamma \leftarrow \langle \boldsymbol{r}, \boldsymbol{r} \rangle$
  repeat
    $\boldsymbol{Hp} \leftarrow$ COMPUTEHP(p)
    $\boldsymbol{s} \leftarrow \boldsymbol{Hp}$
    $\alpha \leftarrow \frac{\gamma}{\langle \boldsymbol{p}, \boldsymbol{s} \rangle}$
    $\boldsymbol{x} \leftarrow \boldsymbol{x} + \alpha \boldsymbol{p}$
    $\boldsymbol{r} \leftarrow \boldsymbol{r} - \alpha \boldsymbol{s}$
    $\kappa \leftarrow \langle \boldsymbol{r}, \boldsymbol{r} \rangle$
    if $\kappa < \epsilon$ **then** ▷ alt. fixed amount of steps
      return
    end if
    $\beta \leftarrow \frac{\kappa}{\gamma}$
    $\boldsymbol{p} \leftarrow \boldsymbol{r} + \beta \boldsymbol{p}$
    $\gamma \leftarrow \kappa$
  until false ▷ exit at 15
end procedure

Computing the inverse is numerically irresponsible. Instead the following linear system is solved $\left(\Delta \boldsymbol{v} = \boldsymbol{v}_i^{(i+1)} - \boldsymbol{v}_i^{(i)}\right)$:

$$\left[\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{v}}\left(\boldsymbol{v}_i^{(i)}\right)\right] \Delta \boldsymbol{v} = \boldsymbol{g}\left(\boldsymbol{v}_i^{(i)}\right). \tag{130}$$

Using the Newton's Method however requires a computation of the Hessian of $E(\boldsymbol{v}_i)$:

$$\frac{\partial \boldsymbol{g}_i}{\partial \boldsymbol{v}_j} = \frac{m_i^n}{4} - \frac{\Delta t^2}{16} \frac{\partial \boldsymbol{f}_i}{\partial \boldsymbol{x}_j}\left(\boldsymbol{x}_i^n + \frac{\Delta t}{4}(\boldsymbol{v}_i + \boldsymbol{v}_i^n)\right). \tag{131}$$

Computing $\frac{\partial f_i}{\partial x_j}$ for every combination ij would be quite memory-intensive. Instead immediately the matrix-vector product on an increment $\delta \boldsymbol{u}_j$ is solved:

$$\sum_j \frac{\partial \boldsymbol{g}_i}{\partial \boldsymbol{v}_j} \delta \boldsymbol{u}_j = \frac{m_i^n}{4} \delta \boldsymbol{u}_j - \frac{\Delta t^2}{16} \sum_j \frac{\partial \boldsymbol{f}_i}{\partial \boldsymbol{x}_j}\left(\boldsymbol{x}_i^n + \frac{\Delta t}{4}(\boldsymbol{v}_i + \boldsymbol{v}_i^n)\right) \delta \boldsymbol{u}_j. \tag{132}$$

$\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{v}}$ is symmetric, positive definite due it being the Hessian of a convex energy function E with a hyper-elastic material. The linear system of eq. 130 can be solved with the conjugate gradient method (Algorithm 1). There is to important things notice:

1. One may want to use a preconditioner $\boldsymbol{P}^{-1}$ to reduce iteration times. However conservation of momentum on incomplete convergence is fulfilled if 130 is premultiplied by the inverse diagonal mass matrix $\boldsymbol{M}^{-1}$ [JST17]. Thus the equation becomes:

$$\left(\frac{1}{4} - \frac{\Delta t^2}{16}\frac{1}{m_i^n}\frac{\partial \boldsymbol{f}_i}{\partial \boldsymbol{x}_j}\left(\boldsymbol{x}_i^n + \frac{\Delta t}{4}(\boldsymbol{v}_i^{(i)} + \boldsymbol{v}_i^n)\right)\right)\Delta v$$

$$= \frac{\boldsymbol{v}_i^{(i)} - \boldsymbol{v}_i^n}{4} - \frac{\Delta t}{4}\frac{1}{m_i^n}\boldsymbol{f}_i\left(\boldsymbol{x}_i^n + \frac{\Delta t}{4}(\boldsymbol{v}_i^{(i)} + \boldsymbol{v}_i^n)\right). \tag{133}$$

2. Due to 132 the product of the Hessian with an increment ($\sum_j \frac{\partial \boldsymbol{f}_i}{\partial \boldsymbol{x}_j}\delta \boldsymbol{u}_j$) needs to be computed anew every step, since a pure matrix store is too costly.

The Hessian is derived following 117 and 118:

$$\frac{\partial e}{\partial x_{i\alpha}\partial x_{j\tau}} = \sum_p V_p^0 \frac{\partial^2 \Psi}{\partial F_{\alpha\beta}\partial F_{\tau\sigma}}(F_p^n)_{\omega\sigma}\frac{\partial w_{jp}}{\partial x_\omega}(F_p^n)_{\gamma\beta}\frac{\partial w_{ip}}{\partial x_\gamma} = -\frac{\partial f_{i\alpha}}{\partial x_{j\tau}}. \tag{134}$$

The Hessian increment can be computed with a two-stage process:

1. Compute a particle quantity $\boldsymbol{A}_p$ with a grid to particle transfer:

$$\boldsymbol{A}_p = \frac{\partial^2 \Psi}{\partial \boldsymbol{F}\partial \boldsymbol{F}}(\hat{\boldsymbol{F}}_p(\hat{\boldsymbol{x}})) : \left(\sum_j \delta \boldsymbol{u}_j (\nabla w_{jp}^n)^T \boldsymbol{F}_p^n\right). \tag{135}$$

2. Compute the Hessian increment ($\delta \boldsymbol{f}_i$) with a particle to grid transfer:

$$-\delta \boldsymbol{f}_i = -\sum_j \frac{\partial \boldsymbol{f}_i}{\partial \boldsymbol{x}_j}\delta \boldsymbol{u}_j = \sum_p V_p^0 \boldsymbol{A}_p (\boldsymbol{F}_p^n)^T \nabla w_{ip}^n. \tag{136}$$

4.7 General-purpose computing on graphics processing units(GPGPU)

4.7.1 GPGPU Optimization opportunities

The speed of these memory types is given in the following starting with the fastest: Register > Shared Memory > Local, Global.

Therefore one should prefer register memory for intra-thread caching. Shared-memory should be used for extra caching memory(to reduce register spilling) or inter-thread communication within a block. Global memory should be mostly used at the start and end of a shader.

Host communication: A common performance bottleneck is stalling the GPU with CPU-GPU transfers. The attainable bandwidth for PCIe x16 3.0 between host and device is roughly 12GB/s. While the communication between device and GPU is 224GB/s (GTX 970). This includes doing calculations on the GPU that may run

Memory	Location on/off chip	Cached	Scope	Lifetime
Register	On	n/a	1 thread(no shuffle)	thread
Local	Off	L2	1 thread	thread
Shared	On	n/a	block	block
Global	Off	L2	all threads+host	host alloc

Table 1: GPU Memory [NVIb]

faster on the CPU. The API may also stall if the CPU is not issuing instructions to the GPU. This may be due to calculations on the CPU, or even worse calculations depending on GPU results. In this thesis the physics already runs on the GPU. Additionally we try to avoid transfers wherever possible.

Memory coalescing: Between a single thread and a block of threads another important entity exists. NVIDIA GPUs group 32 threads into a synchronous warp, respectively AMD groups 64 threads as a wavefront. (Memory) operations are issued per warp/wavefront. A memory access uses the L2 caching behavior. A L2 cache-line is 32 Bytes long. Thus eight (u)int/floats, two single-precision vectors or a single double-precision vector fit into one cache-line. If all threads of a warp access memory within this cache-line only one memory request of 32 bytes needs to be made. In contrast, if a warp accesses N different cache-lines, N*32 bytes need to be loaded.

The unified L1/texture cache acts as a coalescing buffer for threads of a warp (Maxwell architecture [NVId]). It does not matter in which order the cache-line is accessed by the warp. I.e. random or reversed access on a single cache-line are also coalesced. On a global level the easiest to control access pattern thus becomes consecutive or reversed memory accesses. L2 is a write-back cache thus this assessment also qualifies for global memory writes. [Sch]

Shared Memory bank conflicts: Shared memory is split into 32 banks of 4 bytes. The hardware groups together unique memory accesses on these banks. E.g. if two threads within a warp access bank 17 a bank conflict occurs and the access needs to be split into two groups. The one exception is a broadcast: all threads within a warp access the same bank. The number of banks is equal to the warp size, s.t. throughput is maximized if every thread does a unique bank access. [NVIb]

Register spilling: One should avoid local memory. Local memory is created due to register spilling and stored off chip like global memory, though it is L2-cached. The available number of registers per thread is calculable with the CUDA Occupancy calculator [NVIc]. If the occupancy falls off to 0% register spilling occurs. The occupancy is the amount of active warps divided by the possible maximum amount of warps on a Streaming Multiprocessor. It allows the SM to schedule between warps: If the current warp can not continue (memory latency, barrier etc.) it will switch to another; this is called a warp stall, table 2. Occupancy is mostly affected due to resource limits on an SM (shared memory, register) as can be seen in

the occupancy calculator. Non-optimal occupancy might not have any performance impact at all. [NVIa]

To our knowledge there is no way to find out the exact number of registers an OpenGL shader uses. The program object assembly output of *nvemulate* [NVIe] may give an indicator. It will output an intermediate language of one the families of *NV_gpu_programX* (where X stands for the version). TEMP variables are registers. Since *NV_gpu_programX* is a family of intermediate languages there is however no guarentees these are not further optimized. All other metrics discussed are readily available for OpenGL.

Block size: In general the block size should be chosen as small as possible to not hit resource limits. Though we in general had better experiences starting with a block size of 64 as can be seen in the Occupancy calculator. Increasing the block size is useful in combination with shared memory usage. Shared memory access is typically 100x faster than uncached global memory accesses. The more threads of a block reuse shared memory data the faster the memory throughput will be. Therefore operators which involve filtering (also MPM-transfers) can use this behavior.

Multiple elements per thread: Merging threads and simultaneously reducing block size can reduce register pressure if the algorithm allows it. Otherwise merging sequential and parallel work improves performance heavily. Since the block size is limited by the GPU large amounts of data need to be split in tree structures to process in parallel. One can not avoid an indirection of writing-back and loading again from global memory. Therefore algorithms often improve much further from more sequential work to avoid this indirection.

[Aal][NVIb]

4.7.2 GPU Metrics

The GPU metric topping all others is the raw computing time spend on a shader: Δt_c. The following factors matter to set Δt_c into perspective:

1. For the GPU most interesting is the scalability with increasing input data sizes.

2. Numerical algorithms may spend additional time computing (implicit) quantities to allow for larger physical time steps: Δt_p. Therefore one might be interested how long the GPU needs to calculate one physical second:

$$\Delta t^c_{1s} = \frac{\Delta t_c}{\Delta t_p}. \tag{137}$$

   Otherwise one would calculate the speedup of procedure 1 over procedure 2 for a direct comparision:

$$\text{Speedup of Procedure 1} = \frac{\Delta t_{c1}}{\Delta t_{p1}} \frac{\Delta t_{p2}}{\Delta t_{c2}}. \tag{138}$$

Metric	**Max**(GTX 970)	**Description**
Speedup	-	item 2
Δt_c	-	computing time
VRAM SOL%	100	memory througput(, w.r.t. to hw.-limit)
SM SOL%	100	instruction throughput
L2 SOL%	100	L2-cache throughput
Tex SOL%	100	L1-cache throughput
L2-Hitrate%	100	L2 hit-miss ratio
SM Active%	100	at least one warp active avg. over SMs
SM Occupancy	64.0	active warps avg. over SMs
Read Active%	100	% of time spend on VRAM reads
Write Active%	100	% of time spend on VRAM writes
Warp Stall Reasons:		
Long Scoreboard%	100	memory latency
Barrier%	100	block synchronization
Drain%	100	memory wait before warp exit

Table 2: GPU Performance Metrics

The other GPU metrics are acquired using NVIDIA Nsight with OpenGL Performance Markers [NVIg]. An active warp is also one that waits for memory.

None of these metrics necessarily have a correlation to computing time. However they are good indicators on which operations/hardware units the GPU spends its time on. The top SOL% metric measures fraction of the throughput to the bandwidth of that hardware unit. NVIDIA aims to get the top SOL% metric between 60% < top SOL% < 80%. Occupancy was discussed in 4.7.1. [Bav]

Aiming to get shaders memory-bandwidth limited while minimizing the amount of needed accesses for any given workload is the top-priority for our shaders. Achieving memory-bandwidth limit is equal to bringing the VRAM SOL% between 60% < top SOL% < 80%. The metrics Read Active%, Write Active% and Long Scoreboard% will simultaneously increase by following that aim.

5 Implementation

The computing time Δt_c of shaders is gained by OpenGL timer queries [Pie14]. A small benchmarker takes care of queuing timer queries and resolving them a frame later. This is done to not stall the GPU pipeline waiting on those queries. At the end of testing data is gathered and statistical quantities are written into a log file.

5.1 GPU memory layout

Listing 2: AoS Layout

```
1 struct Particle {
2   vec4 position;
3   vec4 velocity;
4 } Particles[n];
```

Listing 3: SoA Layout

```
1 struct Container{
2   vec4 positions[n];
3   vec4 velocities[n];
4 } Particles;
```

Coalescing leads to the motivation of using structure of arrays(SoA) over arrays of structures(AoS) memory layouts. The latter is the more commonly taught approach, while high performance CPU code may also prefer SoA or the more advanced entity component systems. This opens up the field of data-oriented design which is too big too cover and not main part of this thesis.

In short one might not access every variable of a particle in a shader. I.e. in physics one has a pipeline of different passes each requiring another subset of variables. Assume for instance one shader only requires the position then it is more favorable for consecutive threads to load consecutive position vectors. The cache-line is then filled with position vectors, which all will get used. In contrast in an AoS layout the cache-line will have a position and a velocity loaded, but the shader never uses the velocity, halfing the throughput. This process is show in Figure 1 and 2.

Due to this we motivate the C++-library *magic_get* [Pol] for basic reflection. Our code manipulates AoS layouts into SoA layouts and back using reflection. The MSVC-Compiler has a hard coded stack limit for these kind of operations. A struct may be not bigger than 256 Bytes before throwing a compiler error. With the optional use of double-precision this limit is quite restrictive towards that compiler.

Listing 4: OpenGL Layout

```
1  //if SoA
2  #define Particle PREC_VEC_TYPE
3  #define Particle_position 0
4  #define Particle_velocity 1
5  #define Particle_size 2
6  //if AoS
7  struct Particle{
8    PREC_VEC_TYPE position;
9    PREC_VEC_TYPE velocity;
10 };
11 buffer Container{
12   Particle variables [ ];
13 };
```

Furthermore we experienced that for sizes starting with 256 KB the NVIDIA-Compiler could not link the OpenGL program using the SoA Layout as in Listing 3. As an alternative we had to provide meta information in the

style of reflection macros and collapse the container, Listing 4. We can then access SoA by memory offset and AoS by variable or by memory offset for a more abstract view of the memory. We also share the AoS setup between C++ and OpenGL code. One could also build a shader using the abstract layout to convert from SoA to AoS.

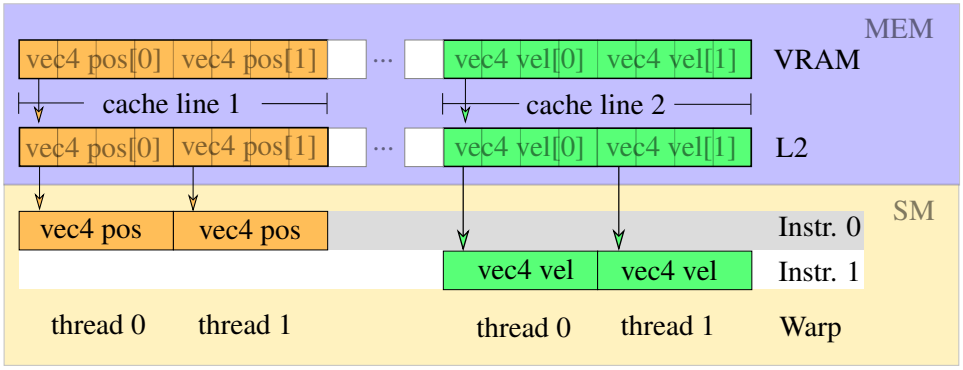

Figure 1: SoA-Layout in practive. The first instruction is shown in orange, the second instruction in green. If green is left out throughput is still maximized

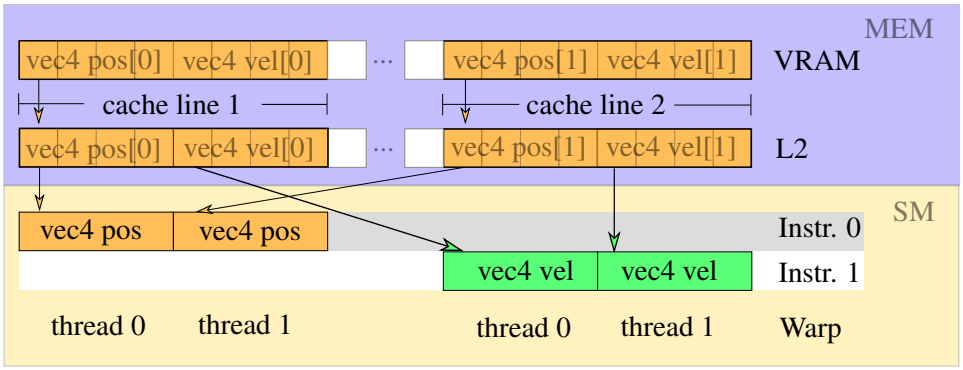

Figure 2: AoS-Layout in practice. The first instruction is shown in orange, the second instruction in green. Orange already loads both cache-lines. If green is left out throughput is halved. Also green is an L2-cache lookup requiring additional overhead over the direct SoA way.

To name a few more cases that macros handle for us:

1. Multiple buffering, e.g. double buffer for unsorted and sorted particles.
2. Single and double precision (PREC_VEC_TYPE in Listing 4).
3. A decorator pattern allows to add buffer information to existing buffer information: For instance index sort needs an extra buffer containing the indices. Full sort requires a double buffer. Thus a sorted buffer gets an attach-

ment(decoration) to contain these information as well as signaling which sort was used in the first place for further operations down the pipeline.

4. Generalize methods to use variants of the same shader with different operations:

```
#define unary_op(x) length(x)
#define binary_op(x,y) x+y
```

For debugging we output the shader with all processed includes and let the gcc-preprocessor assemble the shader with options: `gcc -E -x c -P`. Except for the omission of the glsl version or extension macros, we did not find any problems in this process.

5.2 Parallel Reduction & Scan

The schemes presented in the following are a result of the optimization opportunities discussed in 4.7.1.

Parallel Reduction: Assuming an associative `binary_op(x,y)` := $x \circ y$ and an array of values $[a_0, a_1, ..., a_n]$, parallel reduction computes:

$$r = a_0 \circ a_1 \circ ... \circ a_n. \tag{139}$$

Our OpenGL parallel reduction implementation uses [Har07] as a reference. To avoid memory padding in shared memory the addressing scheme is given by the sequential addressing scheme in Figure 3.

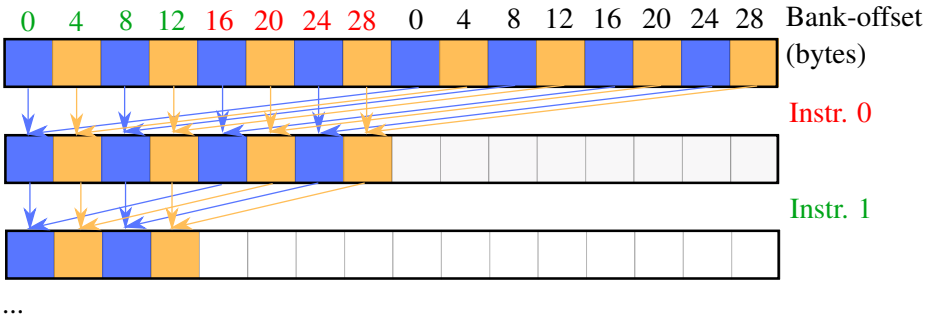

Figure 3: Sequential addressing for `vec4`s. Even threads are blue, odd are yellow. Shared-memory banks per instruction are colored: The banks of the first instruction are green and red numbers. The second instruction is only green. The first and all following instructions can be handled fully concurrently. No memory padding required.

Parallel reduction schemes using shuffle-operations require interleaved addressing due to memory locality restrictions, Figure 5. They use registers instead of shared memory and thus do not need memory padding. [Wes15]

[Har07] also makes heavy use of combining parallel and sequential work. Every thread adds a multiple of two sequential adds into shared memory. After this

the thread does its parallel work. Therefore the reduction factor one invocation does is multiple times higher and allows higher input data sizes without losing any performance, refer to chapter 6.

Parallel reduction solves assuming a `unary_op(x)` directly after memory load:

- CFL-condition of chapter 4.5.4 with operators
  - `unary_op(x) := length(x)` and
  - `binary_op(x,y) := max(x,y)`.
- Discretized conservation of governing equations between time steps or transfers in chapter 4.5.2 and 4.5.3 with
  - `unary_op(x) := x` and
  - `binary_op(x,y) := x + y`.
- Calculate the maximum count of particles of all grid nodes within a block (*per-block parallel reduction* 5.4.3):
  - `unary_op(x) := x` and
  - `binary_op(x,y) := max(x,y)`.

Scan: Assuming an associative `binary_op(x,y)` := $x \circ y$, a neutral element e of the `binary_op` and an array of values $[a_0, a_1, ..., a_n]$, an exclusive scan computes the array:

$$[e, a_0, (a_0 \circ a_1), (a_0 \circ a_1 \circ a_2), \ ... \ , (a_0 \circ a_1 \circ a_2 \circ ... \circ a_{n-1})] . \tag{140}$$

The scan uses an interleaved addressing pattern, Figure 5. The interleaved addressing pattern has the benefit over the sequential one of already computing the partial scans for all odd indices (although misaligned). Note that the reduction needs to reduce to the right achieved by writing into the right element instead of the left in Figure 4. A full scan is then computed as [HSO07]:

1. Upsweep-Phase: Top-to-Bottom reduction to the right of the array.
2. Inserting the neutral element into the last element of the array.
3. Downsweep-Phase: Reverse execution order of item 1 (inverted tree), and an additional right to left element swap.

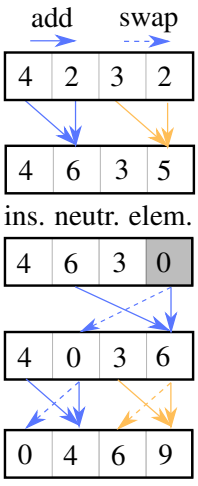

Figure 4: Scan (add)

Again, sequential work can be combined with parallel work. [Bax13] name this process: Raking. Instead of loading one element per thread, one thread loads n elements and computes a sequential partial scan in register memory. The reduced last value of the array is written into shared memory and the parallel scan begins. Afterwards the parallel scan result is spread out with the sequential partial scan.[Bax13] As mentioned, the interleaved addressing requires a memory padding

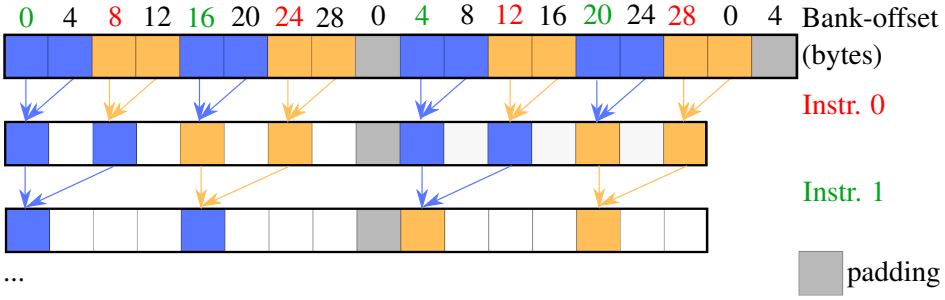

Figure 5: Interleaved addressing `vec4`s. Even threads are blue, odd are yellow. Shared-memory banks per instruction are colored: The banks of the first instruction are green and red numbers. The second instruction is only green. Memory padding needs to be done for concurrent shared-memory access. The first and all following instructions can be handled fully concurrently.

for conflict-free addresses. For block sizes of 32 - 512 this becomes:

$$p = u + (u << \log_2(n)), \text{ or } p = u + \frac{u}{n} \tag{141}$$

where p is the padded memory offset, u is the unpadded memory offset and n is the number of banks. For a block size of 1024 one would need additional padding for a full conflict-free access:

$$p = u + (u << \log_2(n)) + (u << (2 * \log_2(n))), \text{ or } p = u + \frac{u}{n} + \frac{u}{n * n}. \tag{142}$$

However the added computation time does not outweigh the bank-conflicts. So one reverts to eq. 141. Note that this is a correction to [HSO07] where C++ Operator Precedence of $+$ over $<<$ is not recognized and additionally incorrectly $u << n$ is used in the first part of eq. 142.

A scan is used to calculate memory offsets for stream compaction or sorting algorithms.

One can also calculate a per block scan (with halo) without doing a full scan for every block (Appendices B). This implementation did not end up using this more closely related approach to [Hoe14] as it would commit shared memory bank-conflicts defying the architecture of the GPU. Instead preferred is batching a fixed amount of particles, more in chapter 5.4.

The input size for reduction and scan is limited. It's calculable by the $block\ size \times sequential\ loads$. To support bigger input sizes pyramid structures are in use to process the elements the blocks reduced. For our case we mostly reduce the input

size s.t. the first level above in the pyramid can be computed within a single block. For the scan this result is then written back. This was enough for the input sizes we employed and can be similarly extended.

5.3 Counting Sort & Stream Compaction

Sorting on the GPU can dramatically increase workload performance of subsequent steps. If they can profit from following things:

1. Accesses can now be handled in a coalesced fashion.

2. Data can now be reused due to L2-Cache and shared memory.

This is the case for algorithms which require neighbor communication. Supported is either a full deep copy of all particle variables. A deep copy profits from item 1 and 2, but takes the hit in performance of uncoalesced accesses for all variables. Or alternatively an indexing to access an existing particle buffer in the specified sorted manner. Indexing only profits from item 2 and subsequent shaders will be uncoalesced.

Stream Compaction reduces applicable data for subsequent operations. The applicable data is checked by a condition. There are a lot of cases where stream compaction can reduce running times. In chapter 2 particle activation is mentioned as one of such, chapter 5.4.3 will reduce to active blocks. Collision treatment could be implemented as an operation following a collision detection etc. Again indexing and deep copies can be executed for the same reasons mentioned in 1 and 2.

Counting Sort: As the sorting algorithm of choice counting sort is used. Counting sort is split into three parts:

1. **Binning**: Transform the particle position into the grid space, such that every integer in range corresponds to a grid node. Bin due to flooring the transformed particle: Store the particle count c_i of this grid node on the particle as an offset o_p into this grid node.

$$o_p := c_i.$$

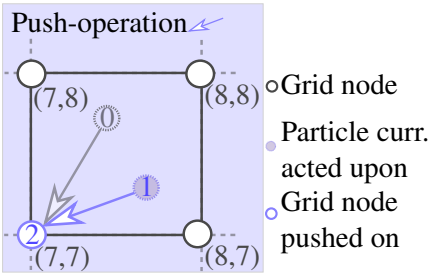

Figure 6: Binning: Particle adds one to c_i after storing it as o_p.

Following that, up the counter on the grid by one:

$$c_i := c_i + 1.$$

This process is visualized in Figure 6.

2. **Scan**: Compute a per grid node scan s_i on the counter c_i following 5.2.

3. **Reordering**: $i_p := s_i + o_p$ will now lead to a new ordering of the particle indices. Sorting is now similar to methods `sort_by_key(uint i)`, where ordering of particles within the same bin is undefined.

[Hoe14][Klá+17]

The one-dimensional indexing of the grid nodes can be chosen as preferred. Possible are a per-block indexing or a global indexing which are both supported.

Particles could potentially access the same bin at the same time resulting in a race condition. A naive solution to the problem are atomic accesses. As long as the (atomic) writes can be handled coalesced this process can be done fast by the GPU, chapter 6. This is only the case for a full deep sort. Therefore binning in the case of a full sort acts upon the last sorted state. Double buffering particles allows to do so. However binning is done after a position update that changes the ordering. Full coalescing will therefore generally not occur.

Alternatively one could work again with the CFL-condition. Since particles move at most one bin in the 27-neighborhood one could mimic a Particle-to-Grid transfer, chapter 5.4.2:

1. **Labeling**: Label the particles by the bin they want to change to.
2. **Neighborhood lookup**: A bin then assigns offsets to the particles in the neighborhood and increases its counter.

Similarly to the Particle-to-Grid transfer one could choose from the different algorithms presented there to do so. However without additional endeavors the order within a bin is not preserved. One would need to rely on the L1-cache as a coalescing buffer as does the atomic global solution.

Alternatively one could choose a closer approach to [Hoe14]'s neighborhood search, which greedily loads all particles. Ordering can easily be preserved then. Note that the support in their approach is smaller (27 instead of 64 nodes) and the data required per particle is only the label.

Chapter 5.4.3 shows how to do this only for active blocks. Due to the difference in support the same pipeline cannot be used.

In summary there is little gains to do so as binning is only executed once per step and is not the dominant performance cost.

5.4 MPM-Operations

Easy to parallelize are all operations that are executed entirely on the grid or entirely on the particles:

$$\square_p = \square_p \circ \square_p \circ ... \circ \square_p,$$
$$\square_i = \square_i \circ \square_i \circ ... \circ \square_i.$$

One thread will just equal to one node or particle respectively. For one variable operations mostly used is a map-shader that applies a unary operation to any input

element. This can be used for instance for the mass divide of momentum on the grid (equation 87) or resetting buffers that are target of global atomic operations.

In contrast much harder are transfers between the two structures, that are essential to the Material-Point-Method:

Particle-to-Grid(P2G)-transfers are the mass (equation 75) and momentum (equation 86) transfers to the grid; as well as the computation of the Hessian of equation 136. In general these can be summarized as:

$$\square_i = \sum_p \square_p \circ \square_{ip}.$$

The transfers back to the particles are called Grid-to-Particle(G2P)-transfers. Apart from the APIC-transfers (equation 90) and the intermediate Matrix $\boldsymbol{A}_p$ for the Hessian (equation 135) the deformation gradient update $\boldsymbol{F}_p^{n+1}$ is also a G2P-transfer (equation 113). In short all transfers of the form:

$$\square_p = \sum_i \square_i \circ \square_{ip}.$$

The number of transfers per frame is much higher than in simple PIC- or SPH-code which are mostly interested in pressure contributions($float$). This motivates to make the transfers a very highly optimized operation. The Material-Point-Method requires for an elastic material as derived:

1. One P2G-Transfer of mass and momentum $m_i, \boldsymbol{v}_i$(combined $float4$).
2. Per Conjugate Gradient Iteration:
   (a) One G2P-Transfer for the 3×3 Matrix $\boldsymbol{A}_p$
   (b) One P2G-Transfer of $\delta \boldsymbol{f}_i$ ($float4$)
3. Per Newton Iteration: One G2P-Transfer for $\boldsymbol{F}_p^{n+1}$
4. One G2P-Transfer of the 3×3 Matrix $\boldsymbol{B}_p^{n+1}$, the position $\boldsymbol{x}_p^{n+1}$ and the velocity $\boldsymbol{v}_p^{n+1}$

In summary this is a lot of data to be transferred between grid and particles, which will only increase with more complex models. In contrast any preprocessing like sorting (chapter 5.3) or filtering active blocks(chapter 5.4.3) need to be executed only once every frame. After that it helps each transfer running more performant. Therefore the transfers are the main focus of this implementation.

The MPM-transfers are different from a typical stencil or filter operation in two main factors:

1. **Dynamic weights**: The stencil is not static. In the MPM-Method the weighting function w_{ip} (or ∇w_{ip} etc.) is dependent on the particle position $w_i(\boldsymbol{x}_p)$. This also means the position needs to be temporarily stored to apply the transfer.

2. **Participating nodes**: A stencil operation reads from a fixed amount of nodes in its neighborhood given by the support of the stencil. This is a node to node relationship which all threads typically participate in from start to finish for the full range of the support.

   In a MPM-Transfer the number of participating nodes can vary. A node may have one, zero or any positive integer of particles associated with it. Thus in a transfer some nodes may not participate at all or have more load than any of the nodes in its neighborhood.

Item 2 does sound more problematic from a parallelization perspective than it is in practice. In chapter 2 particle resampling is mentioned of a means to fill material gaps. This is an inherent mechanic to control the number of particles within a cell. Although it is designed to augment the numerics, the merge method can be easily used to control the upper boundary of particles within a cell. Nevertheless the number of particles within a cell is still variable. Nevertheless all methods presented in the following are not restricted to bins of a fixed size of particles.

The sorting and binning employed in chapter 5.3 allows to split the grid into a partitioning of blocks. Blocks map well to the thread group nature of GPUs. A single thread within a thread group thus corresponds to a grid node within that block.

5.4.1 Grid-to-Particle transfers

The straightforward solution to this problem is to handle all particles independently from each other. One particle adds up the contribution of each grid node variable in range of the interpolation function. In three dimensions this can be done for instance with a two times nested for-loop within a thread. Luckily grid nodes in a uniform grid are inherently sorted which takes care of coalescing.

Furthermore sorting particles due to chapter 5.3 increases caching behavior as variables of grid nodes are directly reused by particles in the same group. This results in direct L2-cache hits instead of going the long way of reloading from global memory.

With the mentioned partitioning of the grid into blocks the shared memory architecture of the GPU can be exploited. This requires two-level binning of particles to nodes within blocks. Figure 7 show how the G2P-transfer using shared memory is realized. Algorithm 5 shows this process briefly. Due to binning a particle is associated with its lower left node. In Figure 7 four threads are active and form the thread group. All visible grid nodes need to be loaded by the threads. This includes the halo of the block, which is dependent upon the support of the interpolation function. For uneven polynomials of the interpolation function the floored position is very favorable. It allows the left support to be one smaller than the right support. As a result the weight of a most outer-edge particle (of the active particle region) just reaches zero for any neighborhood-node outside the halo.

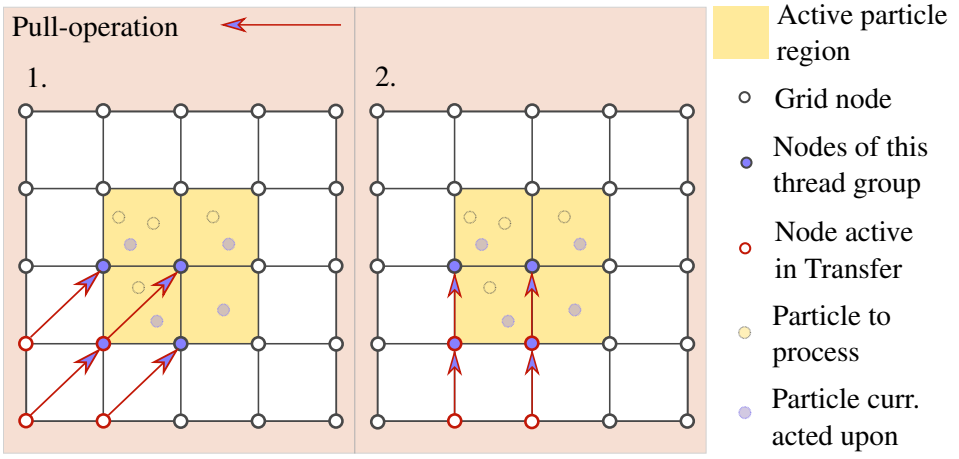

Figure 7: G2P-Transfer: The transfer of a single particle is split into 64 sequential runs (using cubic weights). Shown are the first two runs.

The bigger the block the less halo nodes need to be loaded in total. Given an interpolation function with left support l and right support r, total grid size g and block sizes b_x, b_y, b_z the general function to calculate the number of elements loaded is:

$$m(b_x, b_y, b_z) = \frac{g}{b_x * b_y * b_z}(l + b_x + r)(l + b_y + r)(l + b_z + r). \tag{143}$$

As an example, take a 128*128*128 grid and the cubic interpolation function, which has a left support of 1 and a right support of 2. Subdivide the grid into 4*4*4 or 8*4*4 elements respectively:

$$m(4, 4, 4) : 11,239,424 \text{ Elements}, m(8, 4, 4) : 8,830,976 \text{ Elements}$$

Note however that in practice there is a good change that a major part of the additional elements will result in L2-cache hits.

Next up is the transfer: Each thread corresponds to one particle within the blue nodes in Figure 7. The threads will sequentially run over the neighbors within the support and collect the weighted contribution to this particle. In this process the particle's position need to be loaded to calculate the weight for its corresponding variables. The transfer needs to be repeated for each particle in the bin. Since the number of particles is variable, this loop is dynamic and will lead to branch divergence within a warp. One should rely on external mechanics like particle resampling to reduce branching.

Alternatively one could come up with a per-warp load balancing scheme: One thread would then not directly correspond to one active node. They would need to act as another node in the same warp, when their load is done. This will inevitably lead to shared memory bank conflicts. For the P2G-Transfer this would introduce atomic operations on shared memory again.

Algorithm 5 G2P-Transfer

```
1: for all blockNode ∈ grid do in parallel
2:     shared vec4 blockAndHalo [H_FLAT]
3:     for all (node, local_id) ∈ assignedNodes(blockNode, H_FLAT) do
4:         blockAndHalo[local_id] ← node                          ▷ global load
5:     end for
6:     barrier()
7:     for all particle ∈ bin(blockNode) do
8:         sum ← vec4(.0)
9:         for all neighbor_id ∈ support(blockNode) do            ▷ transfer
10:            w_ip ← weight(particle, neighbor_id)
11:            sum += blockAndHalo[neighbor_id] * w_ip
12:        end for
13:        particle ← sum                                         ▷ global write
14:    end for
15: end for
```

In Appendices C the accesses of one warp on shared memory banks is shown. The interpolation function is assumed to be cubic and the first block dimension is $x = 4$ or $x = 8$. The shared memory can thus be handled as concurrently as possible.

As an extension to this process batching is introduced. Instead of conservatively loading and writing one particle at a time, each thread handles multiple particles together which it stores in register memory. Important to this process is to unroll the corresponding loops that load, write as well as process these multiple particles. Instead of needing to rerun the transfer for each particle, the number of runs now done is the number of non-batched runs divided by the number of particles. This process adds to the amount of registers allocated and can thus also negatively impact performance.

5.4.2 Particle-to-Grid-Transfers

Straightforward to implement are various variations on global memory. Consider first binning was note executed. A grid node would not know how to look up the particles in its neighborhood. The only solution to this is go from the perspective of a particle and write to the grid node in its support. Since any particle in the neighborhood of a grid node contribute to its variables all writes have to be atomic. The approach the could be simply parallelizing for every particle and atomically write to the neighbor nodes in a loop. In the unsorted case this stalls the warps immensely. In [Mey15] the static loop over the support is also parallelized resulting in higher throughput.

The benefits of sorting in chapter 5.3 apply to the P2G-Transfers very significantly. The results of above get reversed and looping performs better. Stalling is

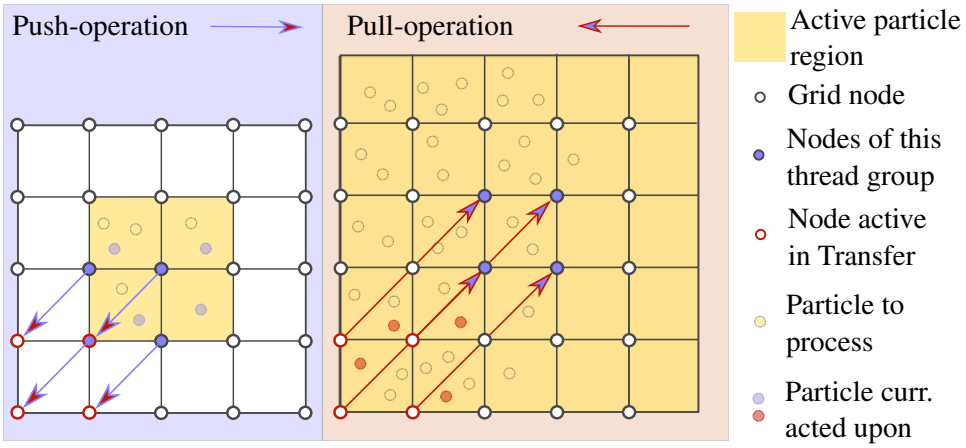

Figure 8: P2G-Transfer: The first run is shown. On the left the P2G-Sync-Transfer is shown that writes into the neighborhood. On the right the P2G-Pull approach is shown that instead reads out of the neighborhood.

much less of an issue.

P2G-Pull-Transfer: However sorting and binning again opens up the possibility of shared memory implementations. The equivalent of the G2P-Transfer for the P2G-Transfer is pulling the relevant data from the neighborhood. We will refer to this as the P2G-Pull-Transfer. The process is shown in Figure 8.

It is important to notice that the supports in the P2G-Pull approach are reversed: Left becomes right, right becomes left. Other than that the transfer works in the same way as the G2P-Transfer Instead of reading from grid nodes, reading from particles is done. Consequently one particle of each bin is loaded into shared memory. This typically puts a lot more strain onto shared memory. For a simple PIC-Transfer(equation 77)

$$(m\boldsymbol{v})_i^n = \sum_p w_i(\boldsymbol{x}_p) m_p \boldsymbol{v}_p^n.$$

the position $\boldsymbol{x}_p$ and the velocity $\boldsymbol{v}_p$ need to be stored in shared memory for every node in the halo. Additionally the count c_i and scan s_i need to be available to load and access particles. For more complex transfers this will get even worse.

By construction the Material-Point-Method tries to avoid matrices on the grid due to storage cost. Conversely the particles have matrices defined over them, e.g. $\boldsymbol{B}_p$,$\boldsymbol{F}_p$. In conclusion one should motivate approaches where grid nodes correspond to shared memory.

There is one implementation detail left out which is solved in chapter 5.4.3. That is grid nodes do not know if there are any particles in the neighborhood left processing. 5.4.3 solves this by finding the maximum particle count in the block (with halo). Thus all nodes within the block need to stay active during transfers. It is possible to delegate this decision down from the block to its subgroups (respecting their respective haloes). Delegating this from the subgroup to the threads

Algorithm 6 P2G-Sync-Transfer

1: **for all** $blockNode \in grid$ **do in parallel**
2: shared vec4 $blockAndHalo$ [H_FLAT]
3: **for all** $local\_id \in$ assignedNodesID($blockNode, H\_FLAT$) **do**
4: $blockAndHalo[local_id] \leftarrow$vec4(.0)
5: **end for**
6: barrier()
7: **for all** $particle \in$ bin($blockNode$) **do** ▷ global load
8: **for all** $neighbor\_id \in$ support($blockNode$) **do** ▷ transfer
9: $w_{ip} \leftarrow weight(particle, neighbor\_id)$
10: $blockAndHalo[neighbor\_id]$ += $particle * w_{ip}$
11: barrier() ▷ uniform flow control needed!
12: **end for**
13: **end for**
14: **for all** $(node, local\_id) \in$ assignedNodes($blockNode, H\_FLAT$) **do**
15: atomicAdd($node$,$blockAndHalo[local\_id]$) ▷ global write
16: **end for**
17: **end for**

is possible as well. Each thread would calculate the summed up count in its neighborhood. But this will hardly matter as all other threads need to wait on the thread with the highest count in the subgroup anyway.

P2G-Atomic-Transfer: One can invert the process of pulling into pushing. Instead of reading from particles this means now writing to grid nodes in shared memory. This immediately alarms one that accesses now need to be either atomic or explicitly scheduled if blocks are larger than a subgroup.

As can be seen in Figure 5.4.2 push approaches have the big advantage of only needing to load particles within the block region. The transfers then spread the particle's contribution to the grid nodes in the neighborhood. If in algorithm 6 line 11 the `barrier()` is left out of the transfer and `atomicAdds()` on shared memory are in use this process is straightforward to implement. One has to make sure to reset the shared memory as their contents are otherwise undefined.

The write-back to memory however now also needs to feature atomics. This process will be the same in the following synchronized approach of line 15 in algorithm 6. For a node that will be part of any halo `atomicAdds()` are a strict requirement. Their results are only partially computed and need to be complemented by neighbor blocks. For such a node these are two writes for a side node, four for an edge node and eight for a corner node.

P2G-Sync-Transfer: Warps/Subgroups are inherently synchronized. No race conditions are present between their threads as they all access different memory positions (as can be seen in Figure 8). The problem arises between different warps of the same thread group. In a synchronized approach this is solved by a `barrier()`. As already mentioned the particle loop of line 7 in algorithm 6 is

dynamic. The OpenGL language makes the following requirements for barriers:

"`barrier()` can be called from flow-control, but it can only be called from uniform flow control. (...) In short, (...) every execution must hit the exact same set of `barrier()` calls in the exact same order." [Khr12]

The dynamic loop has to become static within a group. The problem is solved in same fashion as in the P2G-Pull method with chapter 5.4.3. Calculate the maximum particle count within the block without the halo. This is the minimum amount of times the loop needs to run across the threads to hit uniform flow control.

Warps are still accelerated if all threads within that warp have lower count. They still however have to commit to wait at the same `barrier()`.

In summary all transfers have the same optimal shared memory-access patterns as shown already by Appendices C. Batching is again a valid option that is executed in the same manner by unrolling the corresponding loops. The block size as well as the level of batching can be chosen arbitrarily to optimize for the workload.

5.4.3 Active Blocks & Maximum Block Count:

The P2G-Pull- and P2G-Sync-Transfer of chapter 5.4 required the maximum count of particles within a region. The active particle region is the region that needs to get reduced. Therefore it is sufficient for the P2G-Sync-Transfer to reduce the counter within the block, left of Figure 8. The P2G-Pull-Transfer however also need to take the halo into account, right of Figure 8. Both are achievable with a single level of parallel reduction (in contrast to a pyramid). We will call this the per-block count from now on, that uses as the functional input for the parallel reduction:

1. `unary_op(count) = count.`
2. `binary_op(lc,rc) = max(lc,rc).`

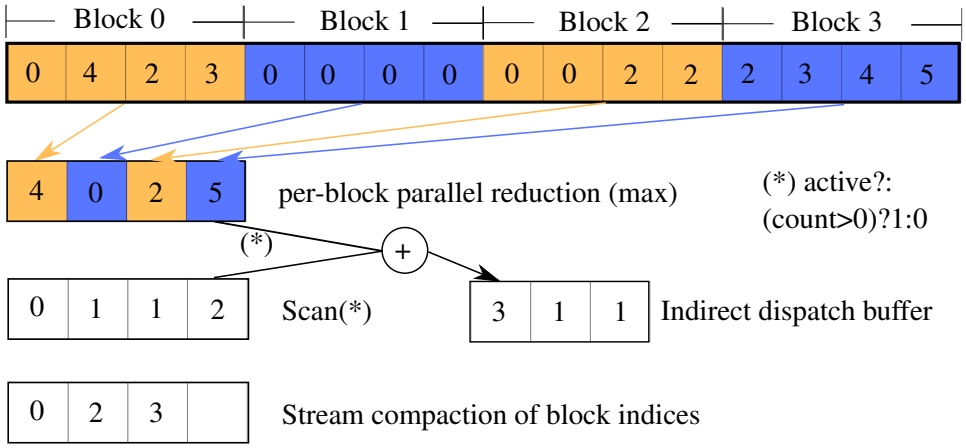

Figure 9: Pipeline to filter active blocks. Returned is the number of active blocks for the launch of a dispatch. A scan and stream compaction gives back the filtered indices to access the active blocks in that dispatch.

For a P2G-Sync-Transfer one may just start with a parallelization level equal to the blocks employed in 5.4. Then can narrow the shader down to do more sequential work to improve throughput. For the P2G-Pull-Transfer one may also do the first, but additionally load all rest elements (by the halo). From there on out more sequential work might be favorable.

If a per block indexing mode is employed all blocks are directly behind each other in the buffer. Reduction then happens in the typical fashion. A global indexing needs to be permuted such that the indexing puts blocks first.

The process of filtering active blocks can be easily appended. Figure 9 shows the process. An active block is one that has a non-zero per-block counter. The scan that will refer to the active block indices takes thus functional input:

1. `unary_op(count) = (count>0)?1:0`, which uses the ternary operator that increases count by one for any active block.

2. `binary_op(lc,rc) = lc + rc`.

Starting a dispatch for every block only the active blocks will now write their index at the scan's position. This process is not coalesced. Coalescing will still be guarenteed for all nodes within the block for the P2G-Transfers and G2P-Transfer. The only non-coalesced access within these shaders is loading the maximum per block count from VRAM, which there are really few of in comparison.

6 Evaluation

6.1 Verifying Results

Parallel reduction, **map** and **scan** are tested against `std::transform_reduce` as it supplies the same functionality as a fully reduced parallel reduction. It is standard since C++17. These shaders are exact on integers. Floating point precision can be seen in table 3.

Counting sort is tested as a whole by giving back the bin keys of the particles. The bin keys are then tested on the CPU against `std::is_sorted`, which is standard since C++11.

Shader	`unary_op`	Total abs. error	Total rel. error
Map (float)	x	0.0	0.0
Map (vec4,dvec4)	length(x)	0.0625	7.94e-08
MapReduce (float)	x	0.0183	3.51e-08
MapReduce (double)	x	1.78e-08	3.42e-14
MapReduce (vec4)	length(x)	212.813	2.71e-04
MapReduce (dvec4)	length(x)	1.58e-05	2.02e-11

Table 3: Error of various shaders on 1024*1024 random elements. Scalars are just between [0.0-1.0]. The vectors have length between [0.0-1.0].

Quantity	Abs. error	Rel. error
m_i^{tot}	212.438	2.12438e-04
$\boldsymbol{p}_i^{tot}$	(0.578,0.787,0.778)	(1.076,1.290,1.204)e-04

Table 4: Error of 1,000 PIC-roundtrips of 1M randomly placed particles with random velocity in a $128 \times 128 \times 128$ grid. Used this instance is a P2G-Sync-Transfer and G2P-Transfer with batching = 4 and block size = (4,4,4). Particles are fixed.

The Eigen-library [JG13] is a well regarded linear algebra library. The **Singular value decomposition** is tested against `Eigen::JacobiSVD`. As a test one million random matrices are created by Eigen. Tests account for the 'Polar SVD'. There is not a single sign error. The average error between the singular values of Eigen and the implementation of chapter 4.2 is 4.91782e-07. $det(\boldsymbol{U})$ and $det(\boldsymbol{V})$ should be as close as possible to 1. The errors are 3.55707e-07 and 3.26429e-07, respectively.

The **MPM-Transfers** need to be mass and momentum conserving. The total mass on the grid m_g^{tot} and the particles m_p^{tot} has to be the same after any transfer:

$$m_p^{tot} = \sum_p m_p = \sum_i m_i = m_g^{tot}. \tag{144}$$

Following this procedure, the same should be true for the total linear momentum $\boldsymbol{p}^{tot}$:

$$\boldsymbol{p}_p^{tot} = \sum_p m_p \boldsymbol{v}_p = \sum_i m_i \boldsymbol{v}_i = \boldsymbol{p}_g^{tot}. \tag{145}$$

And more specifically for the case of APIC also for angular momentum $\boldsymbol{L}^{tot}$ (out of convention upper-case):

$$\boldsymbol{L}_p^{tot} = \sum_p \boldsymbol{x}_p \times m_p \boldsymbol{v}_p = \sum_i \boldsymbol{x}_i \times m_i \boldsymbol{v}_i = \boldsymbol{L}_g^{tot}. \tag{146}$$

An extreme test case spreads one million particles randomly with random velocities on a $128 \times 128 \times 128$ grid, table 4. A roundtrip consists of a P2G-Transfer and a G2P-Transfer for relevant variables. In this process the (random) velocities will get filtered. In PIC-Transfers this regionally averages out velocities. Thus table 4 shows the effects on the total error after 1,000 roundtrips.

As a side note: Random particle positions as well as random velocities are a configuration that in practice will never occur. Even for a fundamental state of matter of gas this is unrealistic.

The total errors vary little between the methods presented in chapter 5.4. They are caused by floating-point inaccuracy. The order of execution on the GPU can vary. Thus the floating-point error will vary from execution to execution.

6.2 Performance

SoA vs. AoS: dvec4 same

Layout	$\Delta t_c(\mu s)$	Speedup	VRAM	SM	L2	Tex
AoS	278	-	77.7%	7.3%	30.3%	14.7%
SoA	122	2.26x	75.4%	14.3%	29.4%	17.9%

Table 5: Timings betweens different buffer layouts. Map operation with one instruction `float x = length(vel)` on 1024×1024 `vec4` buffer storing position and velocity (`vel`)

.

Layout	$\Delta t_c(\mu s)$	Speedup	VRAM	L2	L2-Hit	Read Active
AoS	292	-	61.3%	**41.8**%	53.8%	48.9%
SoA	252	1.16x	**75.4**%	29.4%	20.0%	62.3%

Table 6: Timings betweens different buffer layouts. Map operation with instructions `float x = length(vel), float y = length(pos)` on 1024×1024 `vec4` buffer storing position(`pos`) and velocity (`vel`)

.

Occupancy!

7 Conclusion

7.1 Review

7.2 Future Work

Particle Activation: [Klá+17]
Sparse: [Ope] [NVIf]

Method	$\Delta t_c(\mu s)$	Speedup	VRAM	SM	Top Warp-Stall Reas.
Interleaved	305	-	23.0%	**60.9**%	Short Scoreb.(17.2%)
Sequential	141	2.16x	**49.8**%	37.1%	Long Scoreb.(17.3%)
Seq. (2x)	100	3.05x	69.5%	26.2%	Long Scoreb.(80.1%)
Seq. (128x)	98	3.1x	**72.9**%	16.9%	Long Scoreb.(84.4%)
Seq. (256x)	101	3.0x	66.4%	14.6%	Long Scoreb.(76.9%)

Table 7: Optimization of one parallel reduction dispatch on 1024×1024 instruction `float x= length(vec4 y)`. Note that the interleaved addressing intentionally does bank conflicts to show performance impacts. The number in parentheses states the number of sequential elements each thread loads.

Method	Δt_c	Speedup	VRAM	SM	L2	Top Stall Reas.
Conflicts	748	-	17.5%	51.7%	8.2%	Barrier(49.5%)
No confl.	571	1.31x	25.7%	50.1%	5.4%	Barrier(31.9%)
part. confl.	546	1.36x	23.5%	50.8%	6.0%	Barrier(30.5%)
Seq. (2x)	323	2.31x	41.7%	43.0%	17.2%	S. Scoreb.(32.0%)
unrolled	258	2.90x	53.8%	42.2%	22.0%	S. Scoreb.(32.0%)
Seq. (4x)	311	2.40x	56.8%	22.7%	37.9%	L. Scoreb.(43.3%)
unrolled	297	2.52x	61.1%	21.2%	40.6%	L. Scoreb.(49.9%)

Table 8: Optimization of one scan dispatch on $4 \times 1024 \times 1024$ integers with `binary_op(x,y) := x+y`. Δt_c in μs.

Ordering	$\Delta t_c(\mu s)$	Speedup	VRAM	SM	L2	L2-Hit
Random	1,516	-	25.0%	3.4%	9.1%	10.8%
Deep sort	218	6.95x	75.3%	24.4%	35.0%	37.8%

Table 9: Order dependency of binning of 1024×1024 randomly positioned particles in a $128 \times 128 \times 128$ grid.

Best Case(*)	Δt_c	VRAM	Top Stall Reas.	Read	Write
Index	1,516	25.6%	L. Scoreb.(78.7%)		
Full(8 vec4)	1,731	75.3%	L. Scoreb.(92.2%)		
Worst Case(*)	Δt_c	VRAM	Top Stall Reas.	Read	Write
Index	1,516	26.0%	L. Scoreb.(73.5%)		
Deep Sort(8 vec4)	9,847	24.0%	L. Scoreb.(82.2%)		

Table 10: Order dependency of binning of 1024×1024 randomly positioned particles in a $128 \times 128 \times 128$ grid.

Appendices

A "Inertia Tensor" for Cubic Splines

Algorithm 7: D_p proof

```
import numpy as np
from sympy import *

def round_expr(expr, num_digits):
  return expr.xreplace(
  {n : round(n, num_digits) for n in expr.atoms(Number)}
  )

# Limit a,b,c,x to interval [0,1] for simplyfing
a, b, c = symbols('a b c', nonnegative=true)
a = a/(1+a)
b = b/(1+b)
c = c/(1+c)

x = symbols('x', nonnegative=true)
x = x/(1+x)

# Define cubic interpolation function
def N1(x):
  return 0.5*pow(abs(x),3)-pow(x,2)+2/3
def N2(x):
  return -1/6*pow(abs(x),3)+pow(x,2)-2*abs(x)+4/3
def wip(i,x):
  if(i==1 or i==2):
    return N1(x)
  else:
    return N2(x)

# Define the parametrized position in the grid
def grid_points(x):
  return np.array([-x-1,-x,1-x,2-x])

alphas = grid_points(a)
betas = grid_points(b)
gammas = grid_points(c)

D_temp = np.array([[0,0,0],[0,0,0],[0,0,0]])
```

```
# Loop over all grid nodes in the vicinity
for i,ai in enumerate(alphas):
  for j,bj in enumerate(betas):
    for k,ck in enumerate(gammas):
      # xi_xp is the distance from parametrized
      # position to grid node [i,j,k]
      xi_xp = np.array([ai,bj,ck])
      # each outer product weighted by interpolation functions
      this_outer = wip(i,ai)*wip(j,bj)*wip(k,ck)*
                   np.outer(xi_xp,xi_xp)
      # summed up over all grid nodes
      D_temp = np.add(D_temp,this_outer)

D = np.array([[0.0,0.0,0.0],[0.0,0.0,0.0],[0.0,0.0,0.0]])

for i,D_row in enumerate(D_temp):
  for j,D_ij in enumerate(D_row):
    # simplify to cancel polynoms
    # round_expr because of numerical cancellation
    D[i][j] = round_expr(simplify(D_ij),14)
    print(D)
```

Prints out:

$$\begin{bmatrix} 0.33333333 & 0. & 0. \\ 0. & 0.33333333 & 0. \\ 0. & 0. & 0.33333333 \end{bmatrix} = \frac{\boldsymbol{D}_p}{h^2}$$

B Block Scan

A Block Scan: Assume we have already computed a global scan on a n-dimensional uniform grid and now want to compute the local scan of an n-dimensional tile/block in that grid. This again forms a uniform grid. For simplicity assume two dimensions:
Given the initial two-dimensional pre-scan array:

$$\boldsymbol{G} = \begin{bmatrix} a_{00} & a_{01} & \dots & a_{0n} \\ a_{10} & a_{11} & \dots & a_{1n} \\ \dots & \dots & \dots & \dots \\ a_{m0} & a_{m1} & \dots & a_{mn} \end{bmatrix}. \tag{147}$$

We can identify a block with $i, j, k, l \geq 0$, $i + k < m$ and $j + l < n$ as:

$$\boldsymbol{L} = \begin{bmatrix} a_{ij} & a_{(i)(j+1)} & \cdots & a_{(i)(n-l)} \\ a_{(i+1)(j)} & a_{(i+1)(j+1)} & \cdots & a_{(i+1)(n-l)} \\ \cdots & \cdots & \cdots & \cdots \\ a_{(m-k)(j)} & a_{(m-k)(j+1)} & \cdots & a_{(m-k)(n-l)} \end{bmatrix}. \tag{148}$$

We use the Σ-Notation for sums although any associative binary operator would hold. The global scan is computed as all previous rows + the current row to the current element:

$$S^G_{st} = \sum_{q,r=0}^{s-1,n} a_{qr} + \sum_{r=0}^{(t-1)} a_{sr} \tag{149}$$

with $0 \leq s, t \leq m, n$. The local scan is computed similiar:

$$S^L_{bc} = \sum_{q,r=i,j}^{(b-1),(n-l)} a_{qr} + \sum_{r=j}^{(c-1)} a_{br}. \tag{150}$$

This however requires a full scan of $\boldsymbol{S}^L$, where $i, j \leq b, c \leq (m-k), (n-l)$. We can alternatively compute the local scan from the global scan with the following five steps:

1. Subtract the global scan's first element of every row-element and limit your view to indices b, c then:

$$P_{st} = S^G_{st} - \left(\sum_{q,r=0}^{s-1,n} a_{qr} + \sum_{r=0}^{(j-1)} a_{sr} \right)$$

$$\Rightarrow P_{bc} = \sum_{r=j}^{(c-1)} a_{br} \tag{151}$$

2. Add the last value of $\boldsymbol{L}$ of the previous row (if it exists) to the current row.

$$\Rightarrow T_{bc} = a_{(b-1)(n-l)} + \sum_{r=j}^{(c-1)} a_{br} \tag{152}$$

3. Extract the last column $c = (n-l)$:

$$t_b = a_{(b-1)(n-l)} + \sum_{r=j}^{(n-l-1)} a_{br}. \tag{153}$$

4. Compute the in dimensionality-by-one reduced exclusive scan (of the last column):

$$s_b = \sum_{q=i}^{(b-1)} \left(a_{(q-1)(n-l)} + \sum_{r=j}^{(n-l-1)} a_{qr} \right). \tag{154}$$

5. Row-wise add back:

$$\begin{aligned} T_{bc} + s_b &= \sum_{q=i}^{(b-1)} \left(a_{(q)(n-l)} + \sum_{r=j}^{(n-l-1)} a_{qr} \right) + \sum_{r=j}^{(c-1)} a_{br} \\ &= \sum_{q=i}^{(b-1)} \sum_{r=j}^{(n-l)} a_{qr} + \sum_{r=j}^{(c-1)} a_{br} = S_{bc}^{L}. \end{aligned} \tag{155}$$

For a simpler understanding of this process Figure 10 shows the process with the same numbering:

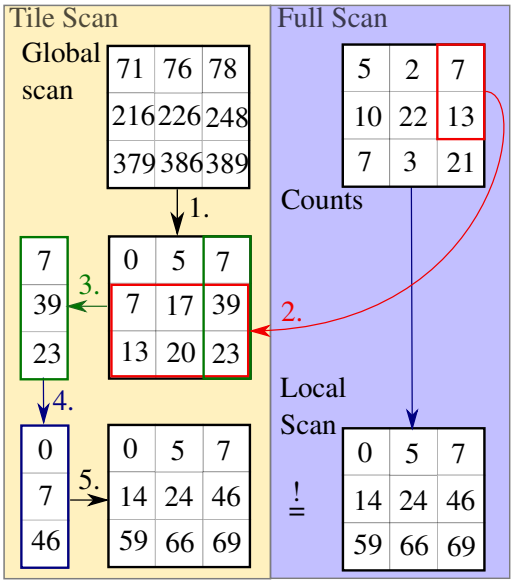

Figure 10: A per block scan following the above enumeration.

C Shared memory accesses on MPM-Transfers

BlockAndHalo(vec4)	0	1	2	3	4	5	6
Banks (1.)	0	1	2	3	4	5	6
Banks (2.)	7	0	1	2	3	4	5
Banks (3.)	6	7	0	1	2	3	4
Banks (4.)	5	6	7	0	1	2	3
Banks (5.)	4	5	6	7	0	1	2
Banks (6.)	3	4	5	6	7	0	1
Banks (7.)	2	3	4	5	6	7	0
Banks (8.)	1	2	3	4	5	6	7
Banks (9.)	0	1	2	3	4	5	6

Table 11: First run of transfer: Bank accesses by warp with block size $x = 4$. Bank accesses are colored grey. The halo ranges due to the support from $0 - 7$. Every bank gets accessed four times. It is easy to see that a window shift would lead to the same amount of bank accesses.

BlockAndHalo(vec4)	0	1	2	3	4	5	6	7	8	9	10
Banks (1.)	0	1	2	3	4	5	6	7	0	1	2
Banks (2.)	3	4	5	6	7	0	1	2	3	4	5
Banks (3.)	6	7	0	1	2	3	4	5	6	7	0
Banks (4.)	1	2	3	4	5	6	7	0	1	2	3
Banks (5.)	4	5	6	7	0	1	2	3	4	5	6

Table 12: First run of transfer: Bank accesses by warp with block size $x = 8$. Bank accesses are colored grey. The halo ranges due to the support from $0 - 10$. Every bank gets accessed four times. It is easy to see that a window shift would lead to the same amount of bank accesses.

List of Figures

List of Tables

Algorithms and source code

References

[Aal] Sebastian Aaltonen. *Optimizing GPU occupancy and resource usage with large thread groups*. https://gpuopen.com/optimizing-gpu-occupancy-resource-usage-large-thread-groups/. [Online; accessed 26-September-2018].

[Abe12] Rohan Abeyaratne. *Volume II of Lecture Notes on,The Mechanics of Elastic Solids: Continuum Mechanics*. http://web.mit.edu/abeyaratne/Volumes/RCA_Vol_II.pdf. [Online; accessed 08-November-2018]. MIT Department of Mechanical Engineering, 2012.

[AER] MIT AEROASTRO. *16.20 Structural Mechanics, Module Notes 3: Constitutive Equations*. http://web.mit.edu/16.20/homepage/3_Constitutive/Constitutive_files/module_3_no_solutions.pdf. [Online; accessed 07-November-2018]. Massachusetts Institute of Technology, Department of Aeronautics and Astronautics.

[Bat06] Klaus-Jürgen Bathe. *Finite element procedures*. Klaus-Jurgen Bathe, 2006.

[Bav] Louis Bavoil. *The Peak-Performance Analysis Method for Optimizing Any GPU Workload*. https://devblogs.nvidia.com/the-peak-performance-analysis-method-for-optimizing-any-gpu-workload. [Online; accessed 15-September-2018]. NVIDIA Corporation.

[Bax13] Sean Baxter. *Reduce and Scan*. https://moderngpu.github.io/scan.html. [Online; accessed 15-October-2018]. NVIDIA Corporation, 2013.

[BP10] Tobias Brandvik and Graham Pullan. "SBLOCK: A framework for efficient stencil-based PDE solvers on multi-core platforms". In: *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*. IEEE. 2010, pp. 1181–1188.

[Bur+10] Heiko Burau et al. "PIConGPU: a fully relativistic particle-in-cell code for a GPU cluster". In: *IEEE Transactions on Plasma Science* 38.10 (2010), pp. 2831–2839.

[Cra+11] Cyril Crassin et al. "Interactive indirect illumination using voxel cone tracing". In: *Computer Graphics Forum*. Vol. 30. 7. Wiley Online Library. 2011, pp. 1921–1930.

[DR08] W. Dahmen and A. Reusken. *Numerik für Ingenieure und Naturwissenschaftler*. Springer-Lehrbuch. Springer Berlin Heidelberg, 2008. ISBN: 9783540764939.

[DS11] Viktor K Decyk and Tajendra V Singh. "Adaptable particle-in-cell algorithms for graphical processing units". In: *Computer Physics Communications* 182.3 (2011), pp. 641–648.

[EHB57] Martha W Evans, Francis H Harlow, and Eleazer Bromberg. *The particle-in-cell method for hydrodynamic calculations*. Tech. rep. LOS ALAMOS NATIONAL LAB NM, 1957.

[Fu+17] Chuyuan Fu et al. "A polynomial particle-in-cell method". In: *ACM Transactions on Graphics (TOG)* 36.6 (2017), p. 222.

[Gao+17] Ming Gao et al. "An adaptive generalized interpolation material point method for simulating elastoplastic materials". In: *ACM Transactions on Graphics (TOG)* 36.6 (2017), p. 223.

[Gas+15] Theodore F Gast et al. "Optimization integrator for large time steps". In: *IEEE transactions on visualization and computer graphics* 21.10 (2015), pp. 1103–1115.

[GM77] Robert A Gingold and Joseph J Monaghan. "Smoothed particle hydrodynamics: theory and application to non-spherical stars". In: *Monthly notices of the royal astronomical society* 181.3 (1977), pp. 375–389.

[Gon00] Oscar Gonzalez. "Exact energy and momentum conserving algorithms for general models in nonlinear elasticity". In: *Computer Methods in Applied Mechanics and Engineering* 190.13-14 (2000), pp. 1763–1783.

[Guo+18] Qi Guo et al. "A Material Point Method for Thin Shells with Frictional Contact". In: *ACM Trans. Graph.* 37.4 (July 2018), pp. 147.1–147.15.

[Har07] Mark Harris. "Optimizing parallel reduction in CUDA". In: *Proc. ACM SIGMOD* 21 (Jan. 2007), pp. 104–110.

[Hoe14] Rama Hoetzlein. *Fast Fixed-Radius Nearest Neighbors: Interactive Million-Particle Fluids*. http://on-demand.gputechconf.com/gtc/2014/presentations/S4117-fast-fixed-radius-nearest-neighbor-gpu.pdf. [Online; accessed 30-September-2018]. NVIDIA Corporation, 2014.

[Hön+] Wolfgang Hönig et al. "A generic approach for developing highly scalable particle-mesh codes for gpus". In:

[HSO07] Mark Harris, Shubhabrata Sengupta, and John D Owens. "Parallel prefix sum (scan) with CUDA". In: *GPU gems* 3.39 (2007), pp. 851–876.

[JG13] Benoît Jacob and Gaël Guennebaud. *Eigen: A C++ linear algebra library*. [Online; accessed 19-January-2019]. 2013.

[Jia+15] Chenfanfu Jiang et al. "The affine particle-in-cell method". In: *ACM Transactions on Graphics (TOG)* 34.4 (2015), p. 51.

[Jia+16] Chenfanfu Jiang et al. "The material point method for simulating continuum materials". In: *ACM SIGGRAPH 2016 Courses*. ACM. 2016, p. 24.

[JST17] Chenfanfu Jiang, Craig Schroeder, and Joseph Teran. “An angular momentum conserving affine-particle-in-cell method”. In: *Journal of Computational Physics* 338 (2017), pp. 137–164.

[KD13] Marcin Krotkiewski and Marcin Dabrowski. “Efficient 3D stencil computations using CUDA”. In: *Parallel Computing* 39.10 (2013), pp. 533–548.

[Khr12] Khronos Group. *OpenGL Wiki: Compute Shader,Shared memory coherency*. [Online; accessed 16-January-2019]. 2012.

[Klá+16] Gergely Klár et al. “Drucker-prager elastoplasticity for sand animation”. In: *ACM Transactions on Graphics (TOG)* 35.4 (2016), p. 103.

[Klá+17] Gergely Klár et al. “Production ready MPM simulations”. In: *ACM SIGGRAPH 2017 Talks*. ACM. 2017, p. 42.

[Kon+11] Xianglong Kong et al. “Particle-in-cell simulations with charge-conserving current deposition on graphic processing units”. In: *Journal of Computational Physics* 230.4 (2011), pp. 1676–1685.

[McA+11a] Aleka McAdams et al. “Computing the singular value decomposition of 3×3 matrices with minimal branching and elementary floating point operations”. In: *Tech. Rep. 1690 University of Wisconsin-Madison* (2011).

[McA+11b] Aleka McAdams et al. “Efficient elasticity for character skinning with contact and collisions”. In: *ACM Transactions on Graphics (TOG)* 30.4 (2011), p. 37.

[McG] Bob McGinty. *Continuum Mechanics*. http://www.continuummechanics.org/basicmath.html. [Online; accessed 08-November-2018].

[Mey15] Fabian Meyer. “Simulation von Schnee”. Bachelor’s Thesis. Institut für Computervisualistik, 2015.

[MIT] MITOPENCOURSEWARE. *18.06 Linear Algebra, Unit III. Lecture Summary: Singular value decomposition*. https://ocw.mit.edu/courses/mathematics/18-06sc-linear-algebra-fall-2011/positive-definite-matrices-and-applications/singular-value-decomposition/MIT18_06SCF11_Ses3.5sum.pdf. [Online; accessed 14-November-2018]. MIT Department of Mechanical Engineering.

[NVIa] NVIDIA Corporation. *Achieved Occupancy*. https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm. [Online; accessed 4-December-2018].

[NVIb] NVIDIA Corporation. *Best Practices Guide - v10.0.130*. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide. [Online; accessed 26-September-2018].

[NVIc] NVIDIA Corporation. *CUDA Occupancy Calculator*. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide. [Online; accessed 3-December-2018].

[NVId] NVIDIA Corporation. *Maxwell Tuning Guide*. https://docs.nvidia.com/cuda/maxwell-tuning-guide/#l1-cache. [Online; accessed 5-December-2018].

[NVIe] NVIDIA Corporation. *NVEmulate*. https://developer.nvidia.com/nvemulate. [Online; accessed 3-December-2018].

[NVIf] NVIDIA Corporation. *NVIDIA GVDB*. https://developer.nvidia.com/gvdb. [Online; accessed 30-September-2018].

[NVIg] NVIDIA Corporation. *NVIDIA Nsight*. https://docs.nvidia.com/nsight-visual-studio-edition/Content/Performance_Markers_OGL.htm. [Online; accessed 5-December-2018].

[Öch14] Andreas Öchsner. *Elasto-plasticity of frame structure elements, Chapter 2: Continuum Mechanics of Plasticity*. Springer, 2014.

[Ope] OpenVDB. *OpenVDB*. https://openvdb.org. [Online; accessed 30-September-2018]. DreamWorks Animation.

[Pay12] Joshua Estes Payne. "Implementation and performance evaluation of a GPU particle-in-cell code". MA thesis. Massachusetts Institute of Technology, 2012.

[Pie14] Daniell Piers. *GL_ARB_timer_query*. https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_timer_query.txt. [Online; accessed 15-September-2018]. The Khronos Group Inc., 2014.

[Pol] Antony Polukhin. *magic$_g$et*. https://github.com/apolukhin/magic_get. [Online; accessed 3-December-2018].

[Ram+15] Daniel Ram et al. "A material point method for viscoelastic fluids, foams and sponges". In: *Proceedings of the 14th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. ACM. 2015, pp. 157–163.

[Ros13] Francesco Rossi. *Particle in cell simulations on GPU clusters*. https://hifweb.lbl.gov/public/BLAST/seminars/2013_05_21/21-05-2013_afrd_Francesco_Rossi.pdf. [Online; accessed 26-September-2018]. 2013.

[Sch] Tim C. Schroeder. *Memory Bandwidth Limited Kernels*. http://developer.download.nvidia.com/CUDA/training/bandwidthlimitedkernels_webinar.pdf. [Online; accessed 3-December-2018].

[SDG08] George Stantchev, William Dorland, and Nail Gumerov. "Fast parallel particle-to-grid interpolation for plasma PIC simulations on the GPU". In: *Journal of Parallel and Distributed Computing* 68.10 (2008), pp. 1339–1349.

[SKB08] Michael Steffen, Robert M Kirby, and Martin Berzins. “Analysis and reduction of quadrature errors in the material point method (MPM)”. In: *International journal for numerical methods in engineering* 76.6 (2008), pp. 922–948.

[Sto+12] Alexey Stomakhin et al. “Energetically consistent invertible elasticity”. In: *Proceedings of the 11th ACM SIGGRAPH/Eurographics conference on Computer Animation*. 2012, pp. 25–32.

[Sto+13] Alexey Stomakhin et al. “A material point method for snow simulation”. In: *ACM Transactions on Graphics (TOG)* 32.4 (2013), p. 102.

[Sto+14] Alexey Stomakhin et al. “Augmented MPM for phase-change and varied materials”. In: *ACM Transactions on Graphics (TOG)* 33.4 (2014), p. 138.

[Str07] G. Strang. *Computational Science and Engineering*. Wellesley-Cambridge Press, 2007. ISBN: 9780961408817.

[Sym] SymPy Development Team. *SymPy: A Python library for symbolic mathematics*. https://www.sympy.org/en/index.html. [Online; accessed 20-November-2018].

[SZS95] Deborah Sulsky, Shi-Jian Zhou, and Howard L Schreyer. “Application of a particle-in-cell method to solid mechanics”. In: *Computer physics communications* 87.1-2 (1995), pp. 236–252.

[Tam+17] Andre Pradhana Tampubolon et al. “Multi-species simulation of porous sand and water mixtures”. In: *ACM Transactions on Graphics (TOG)* 36.4 (2017), p. 105.

[Wes15] Elmar Westphal. *Voting and Shuffling to Optimize Atomic Operations*. https://devblogs.nvidia.com/voting-and-shuffling-optimize-atomic-operations. [Online; accessed 02-October-2018]. NVIDIA Corporation, 2015.

[Wil+07] Samuel Williams et al. “Scientific computing kernels on the cell processor”. In: *International Journal of Parallel Programming* 35.3 (2007), pp. 263–298.