Workshop on Virtual Reality Interaction and Physical Simulation VRIPHYS (2012)
J. Bender, A. Kuijper, D. W. Fellner, and É. Guérin (Editors)

An implicit Tensor-Mass solver on the GPU for soft bodies simulation

X. Faure[1,2] and F. Zara[2] and F. Jaillet[2,3] and J-M. Moreau[2]

[1]Financed by the PRRH (Rhône-Alpes Research Program on Hadrontherapy) for ETOILE (National French Hadrontherapy Centre)
[2]Université de Lyon, CNRS, Université Lyon 1, LIRIS, SAARA team, UMR5205, F-69622, Villeurbanne, France
[3]Université de Lyon, IUT Lyon 1, Computer Science Department, F-01000, Bourg-en-Bresse, France

Abstract

The realistic and interactive simulation of deformable objects has become a challenge in Computer Graphics. In this paper, we propose a GPU implementation of the resolution of the mechanical equations, using a semi-implicit as well as an implicit integration scheme. At the contrary of the classical FEM approach, forces are directly computed at each node of the discretized objects, using the evaluation of the strain energy density of the elements. This approach allows to mix several mechanical behaviors in the same object. Results show a notable speedup of 30, especially in the case of complex scenes. Running times shows that this efficient implementation may contribute to make this model more popular for soft bodies simulations.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Physically based modeling I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation and Virtual Reality I.6.8 [Simulation And Modeling]: Types of Simulation—Parallel

1. Introduction

Following the increasing demand of realism in Computer Graphics, physically-based simulation has become a very active research field over the last decade. This is particularly apparent in cloth and hair animation, medical simulation, interactive entertainment, and more generally in all Virtual Reality applications where animation, interaction or alteration of deformable objects is required in interactive time.

Several methods have been published to model the physical behavior of deformable objects [NMK*06]. Among them, the Finite Element Method (FEM) allows the resolution of the differential equations systems governing the movements of objects [Wor95, BLM00]. This method is based on the discretization of each object into elements (hexahedra or tetrahedra, for example) and provides the resolution of a linear system of the form $KU = F$, with K the stiffness matrix of the considered object, U the displacement of all nodes of the object, and F the external forces and boundary conditions applied on the object. Moreover, K may be dependent of U, judging from the mechanical behavior of the object. As the physical properties of objects are directly integrated in this mechanical formulation, this produces realistic, physically-based simulations of deformable objects.

As an alternative to the classic approach used in the FEM, the Tensor-Mass (TM) approach was introduced in Computer Graphics by Delingette, Cotin and Picinbono [DCA99, CDA00, PDA00, Pic03], and extended by Schwartz [SDR*05]. This approach consists in another way to solve the mechanical equations of the objects, by directly computing the forces applied on each node considering the evaluation of the strain energy density.

To the difference of the classical FEM, the forces between elements do not need to be expressly computed, as only the forces applied on nodes are considered. This approach allows to consider objects discretized on elements with different geometrical models or mechanical properties. The main difficulty remains in the formulation of the equations involved by the forces computation, more specially using an implicit integration scheme like the Euler implicit one, which involves the computation of the differentials of the forces. In our work, this problem is addressed using formal computation.

Several formulations of the Tensor-Mass model have been presented in Computer Graphics to account for various mechanical behaviors: the linear Hookean model [CDA00]; the non-linear geometrical model based on Saint Venant-

DOI: 10.2312/PE/vriphys/vriphys12/001-010

Kirchhoff's elasticity model [PDA00], or anisotropic material [Pic03]. Moreover, Schwartz [SDR*05] presented another extension for non-linear visco-elastic deformations, with some pre-computing of the various tensors to accelerate the process.

Concerning parallel physically-based animation on the GPU, several papers were presented: for non-linear FEM soft tissue modeling based on CUDA [CTAO08], FEM cloth simulation [RNSS*06], implicit FEM solver for deformation simulation [ACF11], or Mass-Springs Systems [MHS05, GW05]. With respect to the Tensor-Mass model, Mosegaard [SM06] addressed its parallelization for the linear case with an explicit integration scheme, using texture memory.

But, as far as we know, the topic of the parallel implementation on the GPU of the Tensor-Mass model has not yet been addressed neither for the non-linear elasticity model, nor using an implicit integration scheme. An approach based on the texture memory as in [SM06] would be possible, but would not permit to consider topological changes in the simulated object's mesh.

In this work, we consider both linear and non-linear elastic models, with two of the most employed formulations in interactive simulations, namely Hookean and Saint Venant-Kirchhoff's models. Moreover, we present the main steps of the resolution of the mechanical equations using the Tensor-Mass approach, employing a semi-implicit as well as an implicit integration scheme to ensure unconditional stability for any time step. The topic of its parallelization on the GPU is also addressed. Obtained results exhibit notable speedup, enabling the interactive simulation of soft-bodies with linear or non-linear mechanical behaviors. Running times shows that this implementation will be valuable for relatively large scenes, contributing to make this model more popular in the domain of deformable object simulation.

2. Simulation of object deformations

Continuum mechanics deals with a continuous way to describe an object moving or being deformed under the action of stress. The deformation Φ may be formulated according to the displacement $U(X)$ of a point X of the object by $\Phi(X) = X + U(X)$.

Different elasticity models exist depending on the expected mechanical behavior of the material. The strain-tensor ε is deduced from the deformation gradient $\nabla\Phi = I + \nabla U$, a quantity depending on ∇U. Therefore, ε may be directly expressed as a function of ∇U. Moreover, the deformation energy W, involved by the deformation, naturally depends on the material's characteristics. Its derivative gives the corresponding opposite force causing the deformation.

Finally, the movement of the deformable object may be expressed by the following differential equations system:

$$M\ddot{U} + D\dot{U} + KU = F, \tag{1}$$

with U the displacement of the object and M, D, K respectively the mass, damping and stiffness matrices of the simulated object.

As for the classical FE approach, the TM formulation is based on the domain's discretization into several elements, and naturally, some steps are similar. However, its specificity is the discrete and local resolution of the mechanical equations. Details of the mains steps involved for each node of each element are now presented.

2.1. Discretization of the displacement in an element

The displacement of a point $X(x,y,z)$ inside an element E of the object is defined by:

$$U_E(X) \simeq \sum_{j=0}^{n-1} \Lambda_j(X)\, U_j, \tag{2}$$

with n the number of nodes $P_j = (P_{jx}, P_{jy}, P_{jz})$ defining the element, U_j the displacement of each node P_j from its initial position, and $\Lambda_j(X)$ some interpolation functions. These functions are defined according to the type of element used for the discretization, with:

$$\sum_{j=0}^{n-1} \Lambda_j(X) = 1. \tag{3}$$

One of the simplest and most used subdivision of the domain consists in *P1 tetrahedral elements* (with $n = 4$ interpolating nodes placed at each vertex). Indeed, this kind of element corresponds to a linear interpolation, that is valid for both considered cases in this paper, namely Hookean and Saint Venant-Kirchhoff's materials, as pointed out in [Pic03].

The linear interpolation functions are defined using the barycentric coordinates of X inside the element, leading, for a tetrahedron, to:

$$\Lambda_j(X) = \alpha_j \cdot X + \beta_j \tag{4}$$

$$= \alpha_{jx}\, x + \alpha_{jy}\, y + \alpha_{jz}\, z + \beta_j \tag{5}$$

for $j = 0..3$ with (considering O the origin and OP_j the position of P_j from this origin):

$$\begin{cases} \alpha_j = (-1)^j (OP_{j+2} - OP_{j+1}) \times (OP_{j+3} - OP_{j+1}) \\ \beta_j = (-1)^j OP_{j+1} \cdot (OP_{j+2} \times OP_{j+3}). \end{cases} \tag{6}$$

2.2. Strain energy for an element

Let us define the strain-tensor and the energy of deformation according to the mechanical behavior of the object.

Hooke's law. For small deformations (or strains) – usually less than 10% of the size of the object – the elasticity is considered as *linear* with a linear relationship between stress and strain. To model such behaviors, only the linear part of the Green-Saint Venant strain-tensor is considered:

$$\varepsilon_l(X) = \frac{1}{2}\left(\nabla U^T(X) + \nabla U(X)\right), \tag{7}$$

with

$$\nabla U(X) = \begin{bmatrix} \frac{\partial U_x(X)}{\partial x} & \frac{\partial U_x(X)}{\partial y} & \frac{\partial U_x(X)}{\partial z} \\ \frac{\partial U_y(X)}{\partial x} & \frac{\partial U_y(X)}{\partial y} & \frac{\partial U_y(X)}{\partial z} \\ \frac{\partial U_z(X)}{\partial x} & \frac{\partial U_z(X)}{\partial y} & \frac{\partial U_z(X)}{\partial z} \end{bmatrix}. \quad (8)$$

The energy density of deformation, measuring the strain energy per unit undeformed volume on an infinitesimal domain around the material point X, is then defined by

$$W_l(X) = \frac{\lambda}{2}(\text{tr}\,\varepsilon_l(X))^2 + \mu\,\text{tr}\,\varepsilon_l(X)^2, \quad (9)$$

where λ and μ are the Lamé coefficients characterizing material stiffness.

Saint-Venant Kirchhoff's constitutive law. For displacements larger than 10% of the object's size, the *non-linear* elasticity behavior must be considered. The simplest hyperelastic model is a direct extension of the linear elastic material. It is based on the Green-Saint Venant strain-tensor ε_{nl}, and its associated strain energy density W_{nl} with:

$$\begin{aligned} \varepsilon_{nl}(X) &= \tfrac{1}{2}\left(\nabla U^T(X) + \nabla U(X) + \nabla U^T(X)\,\nabla U(X)\right) \\ W_{nl}(X) &= \tfrac{\lambda}{2}(\text{tr}\,\varepsilon_{nl}(X))^2 + \mu\,\text{tr}\,\varepsilon_{nl}(X)^2. \end{aligned} \quad (10)$$

Generalization. As ∇U and ∇U^T are constant inside an element when considering P1 tetrahedral elements, the strain-tensor and the deformation energy density (noted ε_{law} and W_{law} according to the chosen mechanical law – Hooke's or Saint Venant-Kirchhoff's) are also constant inside the tetrahedron (*i.e.*, do not depend on X). Hence, the total strain energy W_E of a P1 tetrahedron follows:

$$W_E = \int_E W_{law}(X)\,dX = W_{law}\int_E dX = W_{law}\,Vol_E, \quad (11)$$

with Vol_E the initial volume of the considered element.

2.3. Computation of the forces and their differentials

Considering a given element E of the discretized object, the forces applied on any node P_i for $i \in [0, n-1]$ of this element, is defined by:

$$F_E(P_i) = -\frac{\partial W_E(P_i)}{\partial U_i}, \quad (12)$$

with $W_E(P_i)$ the energy density of deformation of the considered element, evaluated at node P_i.

Whereas the information of the forces is sufficient for an explicit or semi-implicit integration scheme (enabling to compute velocities and displacements according to accelerations and velocities, respectively), the computation of their differentials is mandatory when an implicit integration scheme is used [TPBF87, BW98]. Considering an element E, the differentials of the force on node P_i is given by:

$$\delta F_E(P_i) = \left[\delta F_E^0(P_i)\ \ldots\ \delta F_E^{n-1}(P_i)\right], \quad (13)$$

where $\delta F_E^j(P_i) = \left[\frac{\partial F_E(P_i)}{\partial U_j}\right]_{33}$ for $j \in [0, n-1]$.

Finally, if we consider the whole object involving m nodes, the force applied on $P_\mathcal{I}$ with $\mathcal{I} \in [0, m-1]$ is computed by summing all the contributions from the neighboring elements, with:

$$F(P_\mathcal{I}) = F_\mathcal{I} = \sum_{E \in \mathcal{N}_\mathcal{I}} F_E(P_i) = \sum_{E \in \mathcal{N}_\mathcal{I}} -\frac{\partial W_E(P_i)}{\partial U_i}, \quad (14)$$

where $\mathcal{N}_\mathcal{I}$ is the set of elements containing node $P_\mathcal{I}$, and with $i \in [0, n-1]$ the local node in E corresponding of the global node $\mathcal{I}$ of the object (in the global vertex indexation).

Considering the whole object involving m nodes, the differentials of the force on $P_\mathcal{I}$ with $\mathcal{I} \in [0, m-1]$ follows:

$$\delta F(P_\mathcal{I}) = \delta F_\mathcal{I} = \left[\delta F_\mathcal{I}^0 \ldots\ \delta F_\mathcal{I}^{m-1}\right], \quad (15)$$

with, for $\mathcal{J} \in [0, m-1]$

$$\delta F_\mathcal{I}^\mathcal{J} = \sum_{E \in \mathcal{N}_\mathcal{I}^\mathcal{J}} \delta F_E^j(P_i) = \sum_{E \in \mathcal{N}_\mathcal{I}^\mathcal{J}} \left[\frac{\partial F_E(P_i)}{\partial U_j}\right]_{33},$$

where $\mathcal{N}_\mathcal{I}^\mathcal{J}$ is the set of elements containing global nodes $P_\mathcal{I}$ and $P_\mathcal{J}$, and with i (*resp.* j) $\in [0, n-1]$ the local node in E corresponding of global node $\mathcal{I}$ (*resp.* $\mathcal{J}$) of the object.

2.4. Equation of movement for each node

Once the forces applied on each node have been computed, Newton's equation governing the movement of the object may be used. At time t, considering the node P_i of mass m_i, we have:

$$m_i\,\ddot{U}_i(t) = F_i(t). \quad (16)$$

This equation is linked to the differential equations system (1) by assuming the sparse matrix M to be diagonal, and distributing the object's mass (computed according to its density ρ) on each node of its discretization. This simplification, called mass-lumping, yields to the formulation of equation (16) for each node, which enables the computation of the acceleration according to the applied forces.

The next section will explain in particular our parallel implementation of Euler's semi-implicit and implicit integration schemes used to obtain velocity (from acceleration) and displacement (from velocity) of the nodes throughout simulation.

3. Parallel integration in SOFA

The Open Source Framework SOFA was used for our parallel implementation of the Tensor-Mass model using the OpenCL language [SGS10]. This framework, written in C++ and using the XML script language, enables comparisons between models and methods implemented by research groups in medical simulation (http://www.sofa-framework.org) [ACF*07].

The objects simulated in SOFA are defined by their topology, geometry and visual model (by loading OBJ files, for example), their mechanical state (position, velocity, acceleration and force stored as vectors), their mechanical behavior (constitutive laws), as well as the collision model. Then, the simulation loop is executed, involving the following main steps for each node of the discretized object: (1) computation of the forces, (2) computation of the acceleration according to Newton's second law of motion, (3) integration of the acceleration to obtain velocity, (4) integration of the velocity to obtain displacement.

To integrate the Tensor-Mass model in SOFA, only two functions, namely `addForce()` and `addDforce()`, should be implemented, enabling the computation of the forces and their differentials (only required for Euler's implicit integration scheme) for each node. But, as the initial GPU implementation of SOFA is based on CUDA [CTAO08], the various steps of the simulation loop have also been parallelized using the OpenCL language.

3.1. Forces computation on the GPU

Strategy of parallelization. The same parallelization strategy as the one presented in [ACF11] for an implicit Finite Element solver was adopted in this work, and tailored to our case. Thus, to obtain an efficient parallelization on the GPU, the object's mesh is not partitioned (as this would imply concurrent conflicts), but a two-step process is used instead, involving a parallel computation of forces divided into two tasks, each involving a set of kernels:

1. First, the forces applied on each node of a given element are computed and stored. This computation (`kernel1`) is done for each element of the object.
2. Second, for each node of the object (`kernel2`), these partial forces are summed to obtain the total forces applied on each node (involved by the different elements of the object).

Moreover, to ensure an efficient pooling of these kernels (*i.e.* an efficient use of cores), the number of nodes and elements may be chosen as a power of 2, involving grouping of kernels running in parallel of size a power of 2.

Data structures. Specific data structures were defined for a discretization of the object into N elements involving m nodes, with n the number of nodes of each element and N_n the maximal number of neighbor elements for a node (see Fig. 2-3 for an illustrated example):

- `index` (of size $N \times n$) stores the relationship between local and global indexation for each node of each element. For example, `index[e][v]` gives the global indexation for node `v` of element `e`.
- `PartialForce` (of size $N_n \times m \times 4$) enables the storage of 3D coordinates of partial forces for each node considering its global indexation. Thus, `PartialForce[][v]` gives partial forces of node `v` of the object.
- `TotalForce` (of size $m \times 4$) stores the sum of the partial forces for each node considering its global indexation. Thus, `TotalForce[v]` gives the forces of node `v` of the object.
- `ForceIndex` (of size $N \times n$) stores the index of data structure `PartialForce`. For example, `ForceIndex[e][v]` indicates where the forces applied on vertex `v` involved by the element `e` are stored in `PartialForce`.

To be more efficient on the GPU, one may note that the `float4` data type is used for the storage of partial and total forces. Moreover, basic arithmetic instructions are naturally coded into the hardware (using Arithmetic and Logic Units).

Communication. Fig. 1 illustrates how the global memory is accessed in reading for the initial volume's value of each element i (vol_i in equation (11)). The `float4` data type is again used. And to limit the number of accesses, only one `kernel` over 4 in a grouping of kernels (wrap) will read data from the global memory. In this example, $\texttt{kernel}_0$ reads initial volumes $\{vol_0, vol_1, vol_2, vol_3\}$ and $\texttt{kernel}_4$ reads initial volumes $\{vol_4, vol_5, vol_6, vol_7\}$.

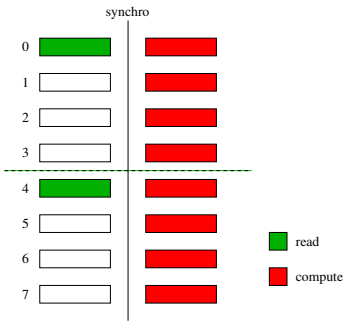


Figure 1: *Only `kernels` of a wrap (grouping of kernels) with a multiple of* 4 *as number will read data from global memory.*

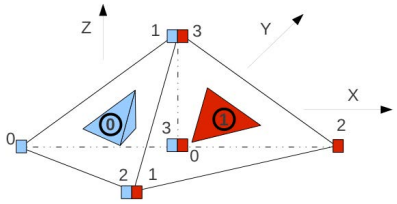
Note that, the same strategy is used to read masses of nodes during the numerical integration scheme.

Parallel algorithm. Algorithm 1 presents the parallel algorithm for force computation – SOFA function `addForce()`. In this algorithm, function `Force()` enables the force computation $F_E(P_i)$ of each node P_i in an element (equation (12)). It depends on constant values (Lamé coefficients λ, μ; coefficients of interpolation functions α_j, β_j for $j = 0..3$ – equation (6)) and some variables (initial position and current displacements of nodes). This function is written using formal computation software and enables to easily generate equations (expressed as polynomials in function of U) of any kind of elasticity models. Consequently, we do not need to construct the matrix formulation of the problem of the form $KU = F$ as for the classical FEM.

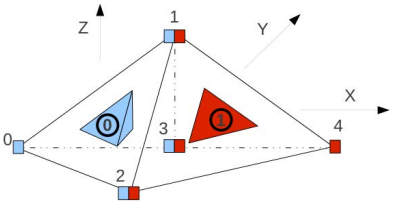
Algorithm 1 addForce() function on the GPU.

```
1: {N : number of elements}
2: {n : number of nodes per element}
3: {m : total number of nodes}
4: {N_n : max number of neighbor elements for a node}
5: // Task 1 - Computation of partial forces
6: for e = 0 to N − 1 do
7:    // Execution of N kernel1
8:    for v = 0 to n − 1 do
9:       PartialForce[ForceIndex[e][v]][index[e][v]]←Force();
10:   end for
11: end for
12: // Task 2 - Sum of partial forces
13: for i = 0 to m − 1 do
14:   // Execution of m kernel2
15:   for j = 0 to N_n − 1 do
16:      TotalForce[i] ←TotalForce[i] + PartialForce[i][j];
17:   end for
18: end for
```

Illustration. To illustrate this parallelization, let us consider a simple deformable object divided into 2 tetrahedra. Fig. 2 exhibits the local and global indexations of nodes. Fig. 3 presents the data structures resulting from this example.



(a) Local vertex indexation.



(b) Global vertex indexation.

Figure 2: *Simple object discretized into* 2 *tetrahedral elements* ($N = 2, n = 4, m = 5, N_n = 2$).

- Consider node P_2 of element E_0, and node P_1 of element E_1. We have `index[0][2] = index[1][1] = 2`. Consequently, node P_2 in global indexation is a common vertex of elements E_0 and E_1.
- Then, the partial forces applied on this node involved by these two elements remain to be computed. Considering element E_0 (*resp.* E_1), `ForceIndex[0][2]` (*resp.* `ForceIndex[1][1]`) containing value 0 (*resp.* 1) indicates where the partial forces computation involved by element E_0 (*resp.* E_1) is stored in `PartialForce`.
- Consequently, `PartialForce[0][2]` (*resp.* `PartialForce[1][2]`) gives the partial forces computation involved by element E_0 (*resp.* E_1). Then, the sum of these two contributions, stored in `TotalForce[2]`, yields the total force applied on node P_2.

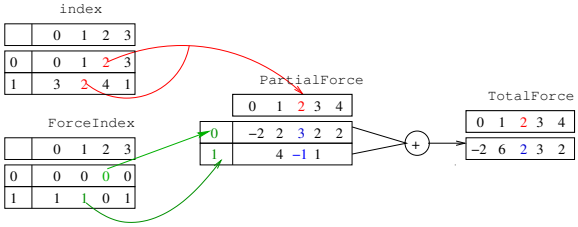


Figure 3: *Data structures for the GPU forces computation.*

3.2. Semi-implicit integration method on the GPU

Once the forces and the acceleration (according Newton's law) are computed, a numerical integration scheme is used to obtain the velocity (according to the acceleration) and displacement (according to the velocity) of the nodes.

Euler's semi-implicit scheme. Firstly, one of the simplest integration methods is considered, namely the *Euler semi-implicit* scheme (also called *Euler symplectic*), defined by:

$$\begin{aligned} V_i(t+h) &= V_i(t) + h\,\dot{V}_i(t) \\ U_i(t+h) &= U_i(t) + h\,V_i(t+h) \end{aligned} \tag{17}$$

where h is the time step and V_i represents the velocity $\dot{U}_i$.

Parallel algorithm. Algorithm 2 presents the parallel algorithm for the computation of the velocity and displacement for each node, using this numerical integration scheme (equation (17)). As previously for the computation of the forces, the `float4` data type is used to store accelerations (`Accel`), velocities (`V`) and displacements (`U`) of nodes.

Algorithm 2 Computation of velocities and displacements on the GPU using Euler's semi-implicit method.

```
1: {m : total number of nodes}
2: // Task 3 - Computation of velocities
3: for i = 0 to m − 1 do
4:    // Execution of m kernel3
5:    V[i] ← V[i] + h * Accel[i] ;
6: end for
7: // Task 4 - Computation of displacements
8: for i = 0 to m − 1 do
9:    // Execution of m kernel4
10:   U[i] ← U[i] + h * V[i] ;
11: end for
```

The parallel implementation of this scheme is straightforward. However, to obtain a stable simulation, the time step h must be small, especially when stiffness increases.

3.3. Implicit integration method on the GPU

Euler's implicit scheme. To enable stability for larger time steps, an implicit integration scheme is mandatory like the *Euler implicit scheme* defined by:

$$\begin{array}{rcl} V_i(t+h) & = & V_i(t)+h\dot{V}_i(t+h), \\ U_i(t+h) & = & U_i(t)+hV_i(t+h). \end{array} \quad (18)$$

Noting M, F, V and U the mass matrix, forces, velocities and displacements vectors, respectively, this scheme may be reformulated in a linear form ($Ax=b$) [TPBF87, BW98]:

$$\underbrace{\left(M-h\frac{\partial F}{\partial V}-h^2\frac{\partial F}{\partial U}\right)}_{A}\underbrace{\Delta V}_{x}=\underbrace{h\,F(t)+h^2\frac{\partial F}{\partial U}V(t)}_{b} \quad (19)$$

with $\Delta V = V(t+h)-V(t)$ and $\frac{\partial F}{\partial U}$, $\frac{\partial F}{\partial V}$ the Jacobian matrices encoding the variation of forces resulting from displacement and velocity changes. Consequently, we have to solve this linear system to obtain ΔV and next update the velocities and the displacements with:

$$\begin{array}{rcl} V(t+h) & = & \Delta V+V(t) \\ U(t+h) & = & U(t)+h\,V(t+h). \end{array} \quad (20)$$

Conjugate Gradient method. The Conjugate Gradient (CG) method, presented in Algorithm 3, is usually favored to solve the linear system (19) in a few iterations [BW98]. In this algorithm, the matrix $\frac{\partial F}{\partial V}$ is null since only internal forces (depending on displacement U) are considered. Thus in this algorithm, only $\frac{\partial F}{\partial U}$ is considered but never explicitly computed, directly computed $h^2\,\frac{\partial F}{\partial U}\,V(t)$ or $h\,\frac{\partial F}{\partial U}\,\Delta V$.

Algorithm 3 Conjugate Gradient algorithm to solve the $Ax=b$ system from Euler's implicit integration scheme.

1: $b \leftarrow h\,F(t)+h^2\frac{\partial F}{\partial U}V(t)$
2: $x \leftarrow 0$
3: $d \leftarrow r \leftarrow b$
4: $\rho_0 \leftarrow < r,r >$
5: **for** i = 1 to n **do**
6: $\quad df \leftarrow -h^2\frac{\partial F}{\partial U}d$
7: $\quad A \leftarrow Md+df$
8: $\quad \alpha \leftarrow \frac{\rho_{i-1}}{<d,A>}$
9: $\quad x \leftarrow x+\alpha\,d$
10: $\quad r \leftarrow r-\alpha\,A$
11: $\quad \rho_i \leftarrow < r,r >$
12: $\quad \beta \leftarrow \frac{\rho_i}{\rho_{i-1}}$
13: $\quad d \leftarrow r+\beta\,d$
14: $\quad$ **if** $(\rho_i > \varepsilon^2\,\rho_0)$
15: $\quad\quad$ **break**
16: **end for**

Parallel algorithms. The parallel computation of differential forces (SOFA's function `addDForce()`) is similar to the parallel computation of forces (SOFA's function `addForce()`) presented in Algorithm 1. The only difference resides in the use of function `DForce()` instead of `Force()` at line 9, to compute differential forces $\delta F_E(P_i)$ of each node P_i in an element (equation (13)). This function depends on the same parameters, in addition to the time step h and the current nodes' velocity, to in fact directly compute $h^2\,\delta F_E(P_i)\,V_E(t)$ or $h\,\delta F_E(P_i)\,\Delta V_E$ (with $V_E(t)$ and ΔV_E the elements of $V(t)$ and ΔV corresponding to E).

For the parallel implementation of the CG method, we used the same parallel strategy as in [ACF11], but using the OpenCL language instead of CUDA. During the execution of this algorithm, only data α, β and ρ are transmitted between GPU and CPU to detect the end of the algorithm.

Finally, the velocities and displacements of nodes are updated in a similar way to Algorithm 2, but replacing the equations (17) by (20).

4. Results

For the simulations, a CPU Intel® Xeon®, 4 cores @3.07 GHz; and a GPU GeForce GTX 560, 2047 MB, 56 cores @1.620 GHz are used. Moreover, a 3D beam of size $5\times1\times1$ m is considered, with a density $\rho = 1{,}000$ Kg.m^{-3} and a Young modulus of 100 MPa.

4.1. Semi-implicit and implicit solvers

Precision. To test the precision of the two numerical integration schemes implemented, a traction test is performed on a Hookean beam discretized into 2,000 parallelepipeds, all split into 6 tetrahedra [MHS05]. A small deformation corresponding to 5 % of the initial beam size is imposed progressively with a displacement of nodes performed with a linear velocity.

Time step	Computed Young's modulus (Pa)	Running time (s)
0.00005	98,775,400	220
0.00010	99,360,500	108
0.00020	100,041,000	52
0.00012	99,392,300	127 (average)

Table 1: *Results for Euler's semi-implicit scheme.*

Time step	Computed Young's modulus (Pa)	Running time (s)
0.00005	96,177,200	841
0.00010	97,839,800	418
0.00020	98,910,300	205
0.00012	97,642,430	488 (average)
Time step	Computed Young's modulus (Pa)	Running time (s)
0.040	99,993,400	5
0.080	99,998,200	2
0.160	99,998,500	1
0.093	99,996,700	3 (average)

Table 2: *Results for Euler's implicit scheme.*

Tables 1 and 2 show the running times to achieve the final state (directly linked to the given time step h) and the Young modulus computed according to the resulting strain. To ensure the stability using the semi-implicit scheme, the largest

time step is $h = 0.0002$, whereas in the implicit case, it is left unconstrained. However, to be complete, tests with the same time steps are presented is the first rows of Table 2.

The precision ratio (initial Young modulus compared to the average of the computed ones) reach 1.00611 (with a time step of 0.00012) and 1.00003 (with a time step of 0.093) for the semi-implicit and the implicit scheme, respectively. Therefore, despite its additional computation cost, an implicit scheme permits to achieve better precision at a reduced final time cost as the time steps may be considerably increased without loosing stability.

Memory. Table 3 shows the memory usage (linear according to the number of elements) and for each data structure, the percentage according to the total required memory. Here, a beam discretized into $N = 60{,}000$ tetrahedra is considered, involving $m = 10{,}000$ nodes, with a max number $N_n = 24$ of neighbor elements for any node. Moreover, an implicit scheme is employed, involving the additional vectors V or ΔV for the semi-implicit one. Note that $P(t_0)$ and $P(t)$ denote the initial and current positions of nodes in this table.

	$P(t_0)$	$P(t)$	V ou ΔV	α_j	Vol_E
Kb	160	160	160	2,880	240
%	1.68	1.68	1.68	30.25	2.52

	PartialForce	index	ForceIndex	TotalForce
Kb	3,840	960	960	160
%	40.34	10.08	10.08	1.68

Table 3: *Memory required for each data structure.*

Only 9.52 Mb are required for our similation, compared to the 560 Mb needed by the Abaqus FE software for the same scene. In the sequel, we will consider only the Euler implicit scheme, which provides very good results, both in precision and time.

4.2. Performances on the GPU

As previously, traction tests are preformed on beams (discretized into 12,000 tetrahedra) with an imposed 5 % deformation of their initial size.

Wrap's size. Fig. 4 presents running times (with time step $h = 0.1$) in function of the wrap's size, *i. e.* the number of cores involved at the same time (Fig. 1).

A stagnation in the execution time appears around 32 kernels in parallel for a total number of 56 cores. Indeed, the number of nodes and elements are chosen as a power of 2, involving an optimal wrap's size equal to a power of 2. In the remaining part of our analysis, we assume the wrap size to be 64, the nearest power of 2 greater than 56.

Number of elements per kernel. In Algorithm 1 for SOFA's function `addForce()`, we chose to execute in *Task*1 one *kernel*1 per element, and we used a similar strategy for SOFA's function `addDForce()`.

Fig. 5 presents the variation of the number of elements treated in *kernel*1 and inside its equivalent kernel in function `addDForce()`, requiring additional loops inside them. The results validate our strategy, *i.e.* to treat only one element per kernel.

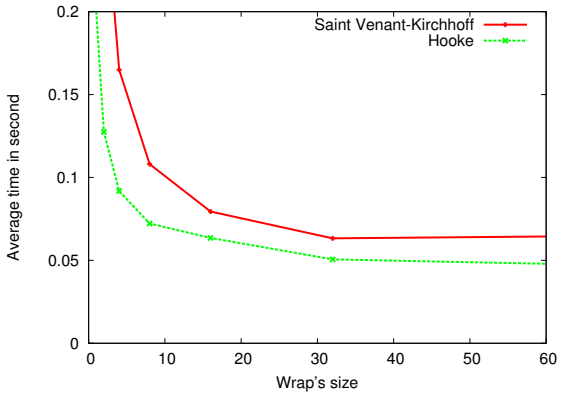


Figure 4: *Average time (in second) of one step as a function of the number of kernels executed in parallel (wrap's size).*

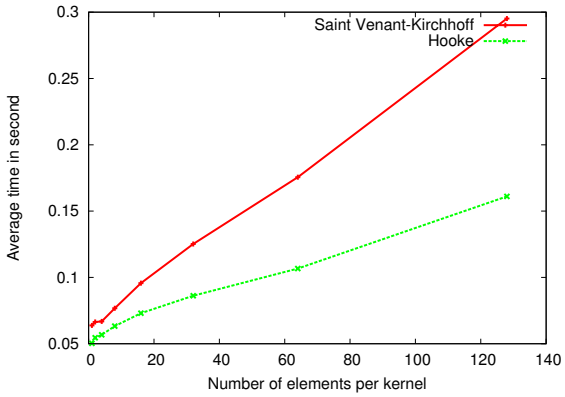


Figure 5: *Average time (in second) of one step as a function of the number of elements per kernel.*

Execution times. Table 4 presents running times (in second) for different parts of the simulation loop, in the case of Hookean and Saint Venant-Kirchhoff's (SVK) elasticity models. For each function, the percentage is given according to the total time. As expected, most of the execution time is consumed by the `addDForce` function.

	addForce	addDForce	CG	Total
Hooke (s)	0.0150	0.1367	0.0059	0.1576
Hooke (%)	9.53	86.75	3.72	100
SVK (s)	0.0147	0.1865	0.0048	0.2060
SVK (%)	7.14	90.53	2.33	100

Table 4: *Execution times (in second) for Hookean and Saint Venant Kirchhoff's elasticity models.*

Speedup. Fig. 6 presents the speedup between the execution times on GPU and CPU. Results obtained with our parallel TM approach, for Hookean and Saint Venant-Kirchhoff's elasticity models, are compared to those obtained using SOFA's FEM (which corresponds to the FEM model presented by Nesme [NP05], that considers non-linear behaviors by computing displacements in a rotated local coordinate system; its GPU version is implemented following [ACF11]).

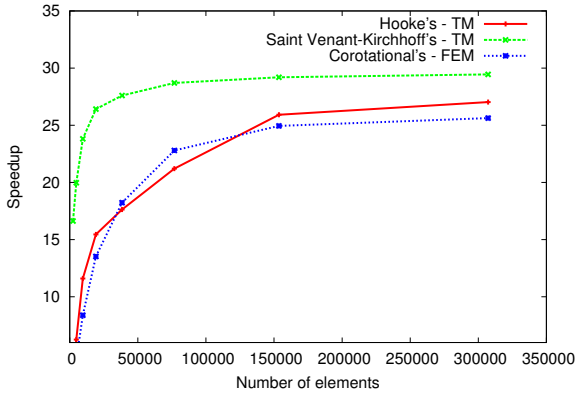


Figure 6: *Speedup between CPU and GPU for TM and SOFA's corotational FEM.*

A speedup of 25.5 is obtained for SOFA's corotational FEM implementation and of 29.5 for our TM, for a Saint Venant-Kirchoff's material beam composed of 307,200 elements.

Consequently, we only used 55 % of the GPU power (with 56 cores), due to synchronization between CPU and GPU in the Conjugate Gradient algorithm. Moreover, we may note that the running time on the CPU is linear according to the number of elements, whereas this time is affine on the GPU with a constant time cost for any number of elements. Therefore, our parallelization on the GPU is interesting for simulations with more than 1,000 elements due to the GPU's cost, judging from the elasticity model used (Hooke or Saint Venant-Kirchhoff).

Data transfer. The data transfer between local and global memory on the GPU is linear and corresponds for each element to constant values (Lamé coefficients λ, μ; coefficients of interpolation functions α_j, β_j for $j = 0..3$ – equation (6)) and some variables (initial position and current displacements of nodes), requiring a total memory of 160 bytes per element.

Scaling study. Fig. 7 presents the computation time for Hookean and Saint Venant-Kirchhoff's elasticity models according to the number of tetrahedra in the mesh. The running times is nearly affine according to the number of elements.

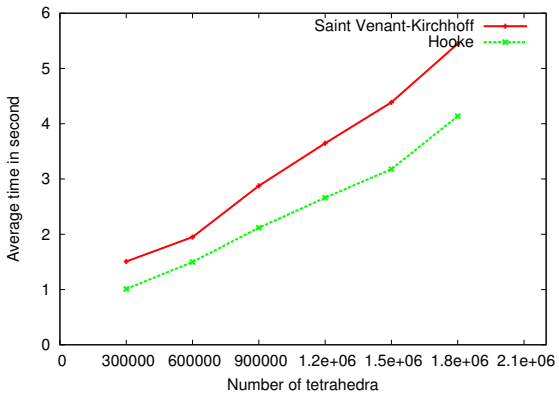


Figure 7: *Average time (in second) of one step as a function of the number of elements in the mesh.*

4.3. Hookean and Saint Venant-Kirchhoff's behavior

Fig. 8 presents a rendered beam for Hookean and Saint Venant-Kirchhoff's material subjected to the gravitational force. Fig. 9 shows a rabbit interactively deformed with Saint Venant-Kirchhoff's elasticity model. Fig. 10 presents some results for torsion and bending of the beam considering both Hookean and Saint Venant-Kirchhoff's materials. In this article, a supplementary color MPEG file is also provided, which contains several views of the 3D simulation for better illustration of our results.

As expected, large deformations involved an increase of the volume for Hooke's elasticity model, whereas Saint Venant-Kirchhoff's material exhibits a much correct behavior.

5. Discussion

These results show that our parallelization of the Tensor-Mass formulation provided very positive results. Its efficiency is increased for large simulations including more than 1,000 elements, that is due to the implementation cost on the GPU.

We also note that the use of the Conjugate Gradient algorithm (needed in the implicit scheme), generated some synchronizations between the CPU and the GPU for data transfer. But, as the majority of the running time is dedicated to the `addDForce()` function, this does not really affect overall performances.

Consequently, to increase speedup, we have now to optimize the `addDForce()` function, and more specifically the `DForce()` function called inside it, which computes the differentials of the forces. As we ever said, this computation is based on formal computation to easily obtain their formulation and to generate the corresponding code. This is one of the reasons why a better speedup is obtained than for the corotational SOFA's FEM (see Fig. 6).

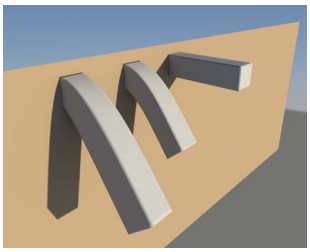
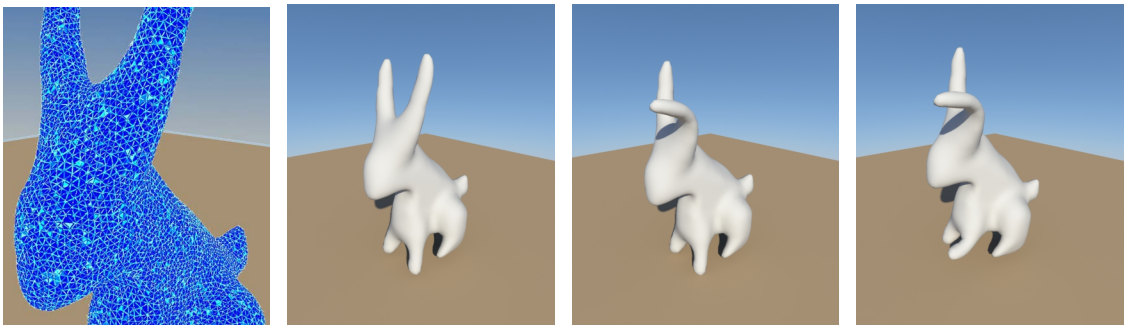
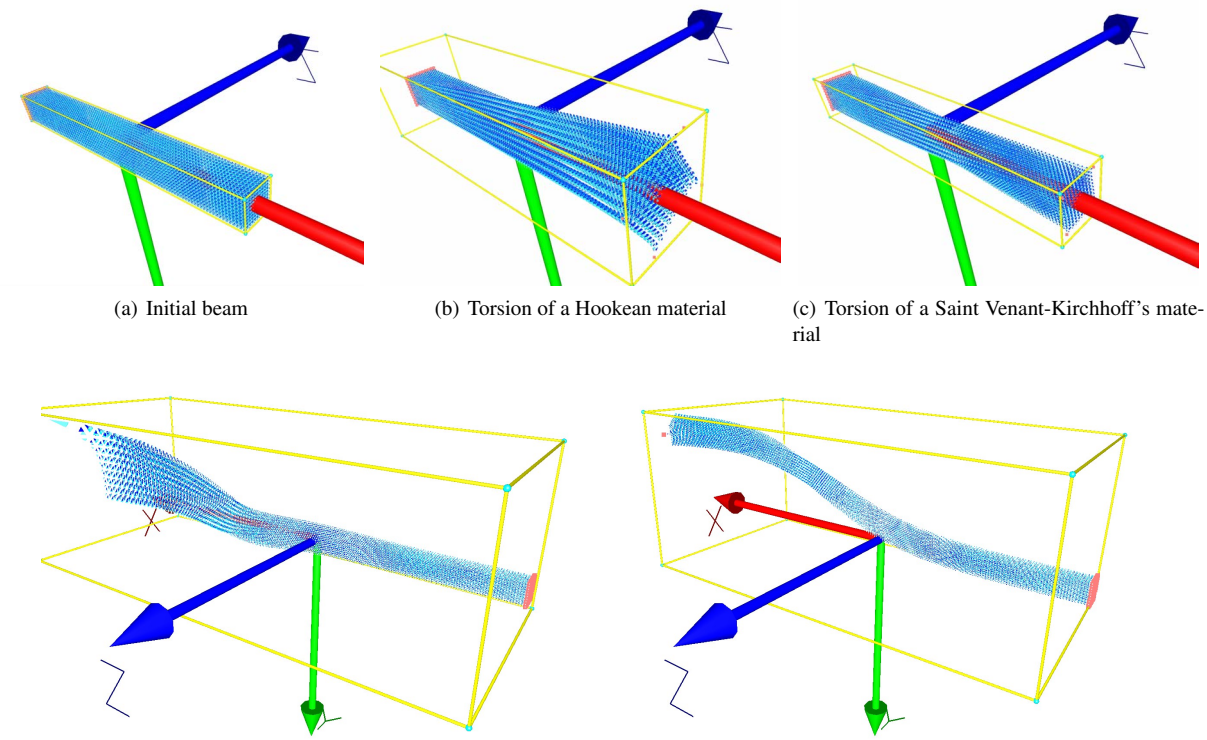


Figure 8: *Rendered beam for Hookean and Saint Venant-Kirchhoff's material, and its initial state (from left to right).*



(a) Initial mesh (zoom) (b) Initial undeformed rabbit (c) Curved left ear (d) Little jump

Figure 9: *Simulation of the deformation of a Saint Venant-Kirchhoff's material rabbit (initial 3D mesh courtesy of L. Stanculescu, with radius-edge ratio $q < 1.0$).*



(a) Initial beam (b) Torsion of a Hookean material (c) Torsion of a Saint Venant-Kirchhoff's material

(d) Bending of a Hookean material (e) Bending of a Saint Venant-Kirchhoff's material

Figure 10: *Hookean versus Saint Venant-Kirchhoff's behavior for torsion and bending test an a beam.*

6. Conclusion

In this paper, we presented an implementation of the Tensor-Mass model on the GPU that considerably speeds up the simulation times. Comparisons between running times on the CPU and the GPU suggest that the parallel implementation of the model becomes interesting for increasingly complex computations.

Besides, we used a formal encoding for the equations of the forces (function `Force()`). However, this allows us to generalize the technique to almost any existing combination of constitutive law and mesh type, without extensive additional coding. This is particularly interesting as it will allow to implement and perform easily a variety of tests and comparisons for various materials on different solvers for an object within the same framework (SOFA, for example).

Moreover, the differentials of the forces are also be obtained formally (function `DForce()`), providing an easy way to implement Euler's implicit integration scheme, which requires these differentials. This computation would otherwise be very fastidious, as the energy equation must be derived twice; and finally, an implicit solver had never been implemented before for the Tensor-Mass model.

In addition, results are presented for tetrahedral meshes, but the derivation for other element types is straightforward, as long as interpolation functions Λ exist for each type (for example: triangle or quadrangle in 2D, and quadratic or cubic tetrahedron or hexahedron, prism, etc. in 3D). The only issue is that W_E in equation (11) is no more constant over the element, necessitating computation that may be done formally as suggested above.

Consequently, we believe that a GPU version of the TM approach will help to make this model popular in the Computer Graphics community. Our implicit parallel solver for linear and non-linear behavior yields a stable version with high time reduction, in comparison with the explicit CPU implementation of the linear TM that may be found in the literature. Moreover, relatively similar simulation times are achieved in comparison to other existing FEM, but allowing better interaction control. This leads to an efficient alternative implementation for interactive physically-based simulation. Indeed, it combines advantages of discrete models (like Mass-Spring Systems) such as topological changes and interactions with the surrounding scene, and those of FEM, as it is derived as well from continuum mechanics.

References

[ACF*07] ALLARD J., COTIN S., FAURE F., BENSOUSSAN P.-J., POYER F., DURIEZ C., DELINGETTE H., GRISONI L.: Sofa an open source framework for medical simulation. In MMVR'15 (Long Beach, USA, February 2007). 3

[ACF11] ALLARD J., COURTECUISSE H., FAURE F.: Implicit FEM Solver on GPU for Interactive Deformation Simulation. In GPU Computing Gems Jade Edition. NVIDIA/Elsevier, Sept. 2011, ch. 21. 2, 4, 6, 8

[BLM00] BELYTSCHKO T., LIU W., MORAN B.: Nonlinear finite elements for continua and structures, vol. 1. Wiley New York, 2000. 1

[BW98] BARAFF D., WITKIN A.: Large steps in cloth simulation. In Proc. of the 25th annual conference on Computer Graphics and Interactive Techniques (1998), ACM, pp. 43–54. 3, 6

[CDA00] COTIN S., DELINGETTE H., AYACHE N.: A hybrid elastic model for real-time cutting, deformations, and force feedback for surgery training and simulation. The Visual Computer 16, 8 (2000), 437–452. 1

[CTAO08] COMAS O., TAYLOR Z., ALLARD J., OURSELIN S.: Efficient Nonlinear FEM for Soft Tissue Modelling and Its GPU Implementation within the Open Source Framework SOFA. In ISBMS 2008, London, UK, July 7-8, 2008: proceedings (2008), vol. 5104, Springer-Verlag New York Inc, p. 28. 2, 4

[DCA99] DELINGETTE H., COTIN S., AYACHE N.: A Hybrid Elastic Model Allowing Real-Time Cutting Deformations and Force Feedback for Surgery Training and Simulation. In Computer Animation (Computer Animation'99) (no address, États-Unis, 1999), Thalmann N., Thalmann D., (Eds.), IEEE Computer Society, pp. 70–81. 1

[GW05] GEORGII J., WESTERMANN R.: Mass-spring systems on the GPU. Simulation Modelling Practice and Theory 13, 8 (2005), 693–702. 2

[MHS05] MOSEGAARD J., HERBORG P., SORENSEN T.: A gpu accelerated spring mass system for surgical simulation. Studies in health technology and informatics 111 (2005), 342–348. 2, 6

[NMK*06] NEALEN A., MÜLLER M., KEISER R., BOXERMAN E., CARLSON M.: Physically Based Deformable Models in Computer Graphics. Computer Graphics Forum 25, 4 (Dec. 2006), 809–836. 1

[NP05] NESME M., PAYAN Y.: Efficient, physically plausible finite elements. Eurographics (short papers) (2005), 1–4. 8

[PDA00] PICINBONO G., DELINGETTE H., AYACHE N.: Real-Time Large Displacement Elasticity for Surgery Simulation: Non-linear Tensor-Mass Model. In Proceedings of MICCAI'00 (London, UK, 2000), Springer-Verlag, pp. 643–652. 1, 2

[Pic03] PICINBONO G.: Non-linear anisotropic elasticity for real-time surgery simulation. Graphical Models 65, 5 (Sept. 2003), 305–321. 1, 2

[RNSS*06] RODRIGUEZ-NAVARRO J., SUSÍN SÁNCHEZ A., ET AL.: Non structured meshes for Cloth GPU simulation using FEM. In VriPhys 2006 (2006). 2

[SDR*05] SCHWARTZ J., DENNINGER M., RANCOURT D., MOISAN C., LAURENDEAU D.: Modelling liver tissue properties using a non-linear visco-elastic model for surgery simulation. Medical Image Analysis 9, 2 (2005), 103–112. 1, 2

[SGS10] STONE J., GOHARA D., SHI G.: Opencl: A parallel programming standard for heterogeneous computing systems. Computing in science & engineering 12, 3 (2010), 66. 3

[SM06] SØRENSEN T., MOSEGAARD J.: An introduction to gpu accelerated surgical simulation. Biomedical Simulation (2006), 93–104. 2

[TPBF87] TERZOPOULOS D., PLATT J., BARR A., FLEISCHER K.: Elastically deformable models. ACM Siggraph Computer Graphics 21, 4 (1987), 205–214. 3, 6

[Wor95] WORKS P.: Finite element modeling for stress analysis. John Wiley & Sons, 1995. 1