An explicit dynamics GPU structural solver for thin shell finite elements

http://dx.doi.org/10.1016/j.compstruc.2015.03.005

A. Bartezzaghi[a], M. Cremonesi[b,*], N. Parolini[c], U. Perego[b]

[a] *CMCS, MATHICSE, École Polytechnique Fédérale de Lausanne, Switzerland*
[b] *Department of Civil and Enviromental Engineering, Politecnico di Milano, Milano, Italy*
[c] *MOX, Department of Mathematics, Politecnico di Milano, Milano, Italy*

Abstract

With the availability of user oriented software tools, dedicated architectures, such as the parallel computing platform and programming model CUDA (Compute Unified Device Architecture) released by NVIDIA, one of the main producers of graphics cards, and of improved, highly performing GPU (Graphics Processing Unit) boards, GPGPU (General Purpose programming on GPU) is attracting increasing interest in the engineering community, for the development of analysis tools suitable to be used in validation/verification and virtual reality applications. For their inherent explicit and decoupled structure, explicit dynamics finite element formulations appear to be particularly attractive for implementations on hybrid CPU/GPU or pure GPU architectures. The issue of an optimized, double-precision finite element GPU implementation of an explicit dynamics finite element solver for elastic shell problems in small strains and large displacements and rotations, using unstructured meshes, is here addressed. The conceptual difference between a GPU implementation directly adapted from a standard CPU approach and a new optimized formulation, specifically conceived for GPUs, is discussed and comparatively assessed. It is shown that a speedup factor of about 5 can be achieved by an optimized algorithm reformulation and careful memory management. A speedup of more than 40 is achieved with respect of state-of-the art commercial codes running on CPU, obtaining real-time simulations in some cases, on commodity hardware. When a last generation GPU board is used, it is shown that a problem with more than 16 millions degrees of freedom can be solved in just few hours of computing time, opening the way to virtualization approaches for real large scale engineering problems.

Keywords: GPU, explicit dynamics, double-precision, shell finite elements.

1. Introduction

In many fields of engineering there is a growing interest for the finite element simulation of large scale problems, involving a great number of unknowns (millions in general), as a replacement for expensive physical testing or prototyping. Even though high-end modern personal computers have multi-core architectures equipped with powerful CPUs, they remain still unable to perform these type of simulations so that the use of supercomputers or clusters is in general necessary.

Graphics Processing Units (GPUs) were originally designed for graphics applications, in particular for video games. The programmable rendering pipeline allowed the highly parallelized execution of small portions of code, but remained relatively difficult to exploit for general purpose computations. Recently, thanks to the development of dedicated hardware with unified architectures and the introduction of specialized programming languages, such as CUDA (Compute Unified Device Architecture), designed by NVIDIA, one of the main producers of graphics cards, or OpenCL, maintained by the Khronos Group and adopted by the major GPU and CPU manufacturers, General Purpose programming on GPUs (GPGPU) has become an increasingly valid computing resource in engineering simulations. Built with thousands of cores, GPUs can drastically increase the computing capacity of the computational nodes at a relatively low cost, even on commodity hardware, opening new scenarios for the development of innovative numerical simulation approaches. In view of this evolution, since several years GPUs have started to be extensively used to speedup calculations in many engineering applications.

A recent review of applications in FEM (Finite Element Method)-based structural mechanics can be found in [1], but GPUs are being used also in a variety of other contexts. In [2], the authors apply high-order Discontinuos Galerkin (DG) method to the solution of Maxwell's equa-

*Corresponding author. Department of Civil and Enviromental Engineering, Politecnico di Milano, piazza Leonardo da Vinci 32, 20133, Milano, Italy

Email addresses: andrea.bartezzaghi@epfl.ch (A. Bartezzaghi), massimiliano.cremonesi@polimi.it (M. Cremonesi), nicola.parolini@polimi.it (N. Parolini), umberto.perego@polimi.it (U. Perego)

Please cite this article as: Bartezzaghi et al. An explicit dynamics GPU structural solver for thin shell finite elements. Comput Struct (2015) http://dx.doi.org/10.1016/j.compstruc.2015.03.005

tions. In DG methods most operators are defined locally at element level. This, combined with the high arithmetic workload and small memory footprint typical of high-order approximations, results in a very favorable environment for GPU implementations. Lattice-Boltzmann methods are also very suited to fine parallelizations and very high speedups can be reached in GPU implementations: see for example [3, 4]. There are also excellent results in molecular dynamics simulations [5], in contact detection for DEM (Discret Element Method) simulations [6] and in boundary element applications [7].

As examples of recent contributions in the FEM context, one can cite the work in [8], concerning the implementation of a domain-decomposition method. With the increasing power of GPUs for general purpose programming, multi-node, multi-core clusters with one (or more) GPU per node have seen a rapid diffusion. The approach presented in [8] shows a hybrid (i.e. based on combined use of CPU and GPU) implementation of the FETI method for implicit finite element structural mechanics problems. The proposed dynamic load balancing algorithm allows for the efficient use of both CPUs and GPUs, minimizing the idle time and exploiting the capabilities of the heterogeneous hardware. An explicit nonlinear finite element solver (which uses linear hexahedra and tetrahedra) for surgical simulations is presented in [9], showing a speedup of more than 20 times compared with a CPU implementation. Always considering surgical applications, [10] proposes GPU-based implementation of the FEM using implicit time integration for dynamic nonlinear deformation analysis. In [11], an explicit dynamics formulation for the simulation of sheet forming problems, based on the use of shell elements of the Belytschko-Tsay type, is discussed. In [12], a new method for the assembling of the stiffness matrix in case of isogeometric analysis is proposed. Hybrid implementations of multiscale finite element approaches, whereby constitutive material computations at Gauss point level are carried out on the GPU, are discussed in [13] and [14]

In most of the above-mentioned applications, results are limited to single-precision arithmetic (e.g. [2, 9, 14, 10]). It is important to recall, as suggested in [8], that, concerning NVIDIA GPUs, in all the implementations before CUDA compute capability 1.3, only single-precision floating point operations are supported directly by the hardware. Furthermore, double-precision variables consume twice as much of the limited amount of GPU registers and block shared memory, and in view of the possible decay of performances implied by the use of device global memory, this has often been considered as a strong indication towards the use of single-precision arithmetic. In addition, the use of single-precision has been a source of misinterpretations in speedup comparisons between CPU and GPU implementations of the same algorithms. GPUs have always been very optimized for high-density single-precision floating point computations (being primarily designed for graphics acceleration); therefore, they can carry out a very large number of Floating Point Operations per Second (FLOPS), if compared to CPUs. However, in most engineering applications double-precision arithmetic is mandatory. Therefore, a certain performance drop when using double-precision instead of single-precision in GPUs has to be accepted as unavoidable (even though on newer GPUs this gap is being reduced, on older ones double-precision could only be emulated and thus performances were very poor).

As pointed out in [1], most structural FEM applications using GPUs are based on implicit algorithms, where GPUs are mainly used for the solution of linear systems of equations. Emphasis is therefore placed on optimizations of the linear solver on GPU. This is the case, for instance, of most general purpose commercial FEM software products (see e.g. [15]). In [1], an accurate comparison of different implicit FEM-based GPU solvers is presented. All the steps of the solution scheme are discussed (pre-processing, solution and post-processing) and the relative speedups are shown. Only implicit schemes are considered as system solving strategy, with performance comparison of various linear system solvers (both direct and iterative). Other extensive studies on the performances of different linear solvers have been carried out for example in [16, 17, 18, 19]. On the other hand, only few GPU implementations of FEM in explicit dynamics are available in the literature. Examples can be found in [9, 11].

In this work, we present a double-precision, explicit dynamics non-linear finite element shell solver on unstructured grids, designed specifically to carry out the whole computation on a GPU. An innovative algorithm implementation of the central difference scheme is proposed in order to fully exploit the computational power of GPUs. Thanks to the original and highly optimized algorithm formulation and careful memory management, the GPU-only implementation is able to reach very high performances with limited memory consumption, even on single-GPU configurations on commodity hardware. For some problems, simulation can also be run at real-time speed.

This paper is structured as follows: in section 2 the finite element structural solver is introduced, with a brief description of the mathematical model and time integration scheme; the GPU implementation, together with the adopted strategies and performance improvements, is described in section 3; numerical results for three test cases are reported in section 4, the third case consisting of a performance analysis on larger meshes; finally, in section 5 limitations of the presented work are discussed, highlighting possible future improvements.

2. Shell structural solver

The main objective of this work is to develop a highly efficient explicit finite element structural solver for the simulation of the dynamical behavior of thin-walled structures. The nonlinear structural problem is discretized by means of *MITC4* shell elements [20, 21], where the typical constraints of Reissner-Mindlin shell kinematics are enforced.

More specifically, MITC4 elements have 4 nodes and 5 Degrees Of Freedom (DOFs) per node, specifically the three displacements and the two rotations around the axes tangent to the element middle plane. The drilling rotation is neglected. In order to avoid shear locking phenomena, a mixed interpolation of the shear stress component is used [22]. The adopted formulation lies within the context of isotropic elasticity, with large displacements and small strains.

2.1. Time integration

Let us consider the semi-discretized equation of motion (for the variational formulation in explicit dynamics leading to the discretized form, see e.g. [23], chapt. 6):

$$\mathbf{M}\ddot{\mathbf{U}} + \mathbf{C}\dot{\mathbf{U}} + \mathbf{F}_I(\mathbf{U}) = \mathbf{F}_E, \tag{2.1}$$

where $\mathbf{U}$, $\dot{\mathbf{U}}$ and $\ddot{\mathbf{U}}$ are vectors containing nodal displacements, velocities and accelerations respectively, $\mathbf{F}_I$ represents the internal equivalent elastic forces and $\mathbf{F}_E$ the external loads. $\mathbf{M}$ is the lumped mass matrix and $\mathbf{C}$ is the damping matrix. The explicit half-station central difference approach [24, 23] is employed to integrate the system of non-linear second order differential equations in (2.1). The damping matrix is assumed to be diagonal and proportional to the mass matrix, i.e. $\mathbf{C} = \alpha\mathbf{M}$ with $\alpha \geq 0$. Together with mass lumping, this choice allows to transform the problem into an uncoupled system of algebraic equations in which each solution component may be computed independently, without assembling any global matrix. Therefore, it is possible to rewrite (2.1) in terms of the individual i-th DOF:

$$m_i a_i + c_i v_i + f_{I,i} = f_{E,i} \tag{2.2}$$

where a_i, v_i and u_i represent the acceleration, velocity and displacement of the i-th DOF respectively. Let us consider a discrete number of time stations $t^n \in [0, T]$, T being the total analysis duration. Let $t^{n+\frac{1}{2}} = \frac{1}{2}(t^{n+1} + t^n)$ define the half-stations between two subsequent time stations. Time steps and half-station time steps are defined as

$$\Delta t^{n+\frac{1}{2}} = t^{n+1} - t^n; \quad \Delta t^n = t^{n+\frac{1}{2}} - t^{n-\frac{1}{2}} \tag{2.3}$$

The following sequence of operations describes the application of the standard central difference time integration algorithm (see [23] for further details):

1. Initialization: set u_i^0, v_i^0 equal to assigned initial conditions, and compute a_i^0 from the initial values of the external forces.
2. Time update:
$$t^{n+1} = t^n + \Delta t^{n+\frac{1}{2}}, \quad t^{n+\frac{1}{2}} = \frac{1}{2}(t^{n+1} + t^n). \tag{2.4}$$
3. First half-step: update values of free DOFs:
$$v_i^{n+\frac{1}{2}} = v_i^n + (t^{n+\frac{1}{2}} - t^n)\, a_i^n, \tag{2.5}$$
$$u_i^{n+1} = u_i^n + \Delta t^{n+\frac{1}{2}}\, v^{n+\frac{1}{2}}. \tag{2.6}$$
and of constrained DOFs. This calculation is carried out independently for each DOF.
4. Loop over all elements:
   - Compute element nodal contributions to internal and external forces: $f_{I,i}^{n+1}, f_{E,i}^{n+1}$.
   - Transform nodal contributions from element local basis to global basis.
5. Second half-step: accelerations and velocities are updated using the new internal and external forces:
$$a_i^{n+1} = \frac{f_{E,i}^{n+1} - f_{I,i}^{n+1} - c_i\, v_i^{n+\frac{1}{2}}}{m_i + (t^{n+1} - t^{n+\frac{1}{2}})c_i}, \tag{2.7}$$
$$v_i^{n+1} = v_i^{n+\frac{1}{2}} + (t^{n+1} - t^{n+\frac{1}{2}})\, a_i^{n+1}. \tag{2.8}$$

The computation of internal and external forces (step 4) is the most expensive step. While steps 3 and 5 are executed independently on each DOF, step 4 involves a loop over the elements and stress and strain tensor calculations require a considerable computational effort. Moreover, after the internal forces are evaluated, they need to be accumulated in per-DOF arrays, since each node receives force contributions from all the elements that share it.
The main disadvantage of explicit time integration schemes consists of their conditional stability. A simple criterion to determine a stable time step for simulation on an assigned mesh is given by ([23]):

$$\Delta t = \gamma\, \Delta t^{crit}; \quad \Delta t^{crit} = \frac{2}{\omega_{max}} \leq \min_e \frac{l_e}{\sqrt{\frac{E}{\rho(1-\nu^2)}}}, \tag{2.9}$$

where ω_{max} is the highest frequency of the assembled linearized system, l_e is element e characteristic length, ρ the density, E the Young modulus, ν the Poisson ratio and γ is a reduction factor accounting for the problem nonlinearity. This limitation usually leads to very small time steps, which makes the method particularly suited to high strain rate dynamics problems. Nevertheless, explicit schemes are often conveniently used also in relatively slow dynamical applications as long as the problem is highly nonlinear, since they do not require iterations to reach convergence within a time step.

3. GPU Implementation

The explicit algorithm presented in the previous section has been developed for execution on NVIDIA GPUs using CUDA, which was chosen among the available GPGPU frameworks for its stability and maturity. For a description of CUDA and NVIDIA GPUs' architecture see [25]. However, the majority of the concepts described in the following sections are general and can be exploited on other platforms, with some minor changes. The implementation presented here is meant for GPUs with CUDA compute capability 2.0, with the developed code optimized

for that architecture: older devices are not supported and some features of newer GPUs may not be exploited. In addition, the solver currently runs on single-GPU machines, even though extending the current implementation to a multi-GPU environment is straightforward. A basic graphical representation of the different elements of the GPU architecture, together with the relevant terminology used in GPU programming, is provided in figure 1. A generic CUDA application is mainly divided into two parts: *host* code, running serially or in parallel on CPU, and *device* code, running on one (or more) GPUs. The host code is responsible for initializing the GPU device and organizing memory and managing transfers. The device code usually provides a set of functions callable from the CPU code, named *kernels*. A kernel is basically a sequence of *SPMD* operations (Single Program Multiple Data) executed on the GPU on an arbitrary number of parallel threads. The threads are organized in n-dimensional *blocks*, which are then organized in a n-dimensional *grid* (where n can be 1, 2 or 3). A GPU contains a certain number of *Streaming Multiprocessors* (SM), which execute one or more thread block each, based on the resource available and requested per-block. Threads are then executed by the SM in groups of 32, called *warps*, with virtually zero-overhead warp scheduling, handled by the hardware transparently to the user. Threads can be synchronized explicitly only inside the blocks; communications between different blocks is more complicated and costly, as it has to be done using global memory fences and atomic operations. Therefore, the organization of the threads is an important aspect to be kept in consideration when developing in CUDA. For a more detailed description of the programming model, the types of memories available and other considerations regarding CUDA, see [25].
If we consider a single-CPU host machine with a single GPU board, we can generically subdivide a GPGPU application in four essential macro-steps:

- initialization and organization of permanent data and constants in the memory on the GPU board;
- transfer of input values from the host (CPU) machine to the device (GPU);
- execution of one or more GPU kernels on this data;
- transfer of output values and results from the device to the host machine.

For simple problems, porting to GPU an existing algorithm developed for a CPU can be straightforward, reflecting the actual CPU implementation. However, for more complex tasks, to achieve good performances and take full advantage of the GPU computational power, the design of algorithms should reflect the actual structure of the GPU. More specifically, particular care must be taken in organizing memory and managing data transfers. GPUs are in fact designed to be very efficient at performing floating point arithmetic, but are not optimized to cope with branches and execution divergences, or random device memory accesses. When writing GPU code, the difference in performance obtained accessing device memory aligned and with specific patterns rather than with random access is substantial. In addition, because of the limited *PCI-Express* bandwidth, host-to-device and device-to-host memory transfers can easily become the bottleneck of an algorithm. Therefore, constant data should be kept on GPU memory and moved from/to the device only when necessary. In this work, we assume all data (computational mesh and other required data) to fit within the available GPU memory. While this can be seen as a restriction, it is worth noticing that the actual memory consumption of the presented implementation is very low, allowing for simulation of realistic engineering problems with a large amount of degrees of freedom even on a single GPU board.
In the next sections, two different GPU implementations of the algorithm described in section 2.1 are presented and critically discussed. The first one consists in a straightforward implementation of the CPU algorithm, slightly adapted to take advantage of the GPU parallel characteristics. In the second case, the algorithm is reconsidered from scratch, to achieve an optimized GPU-only implementation.

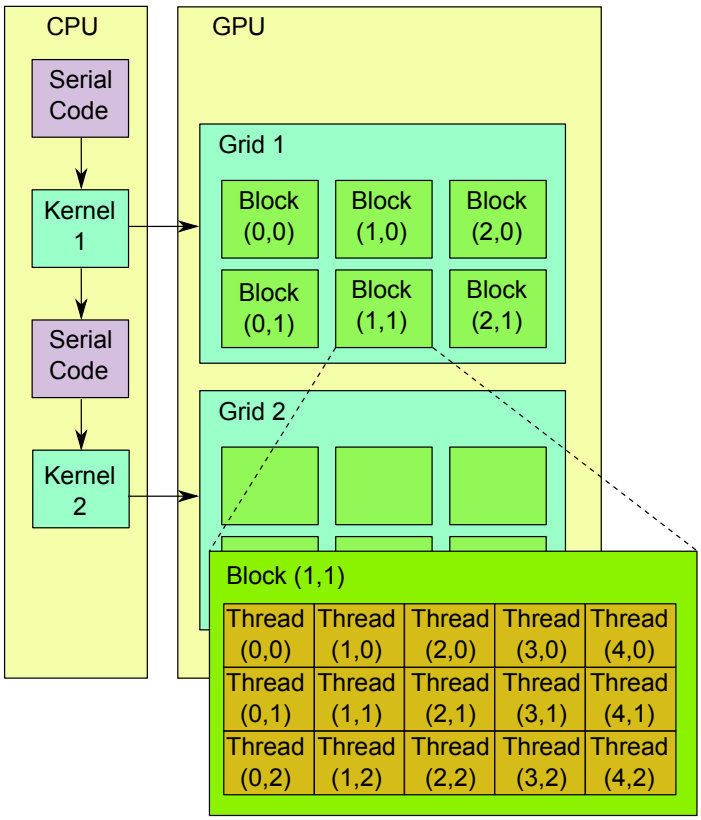


Figure 1: Schematic illustration of GPU architecture and programming elements.

3.1. First GPU implementation

The first GPU implementation considered is adapted directly from the serial CPU algorithm. The various steps of the time advancing scheme described in section in 2.1 are implemented in different kernels (figure 1): those concerning the two integration half-steps (steps 3 and 5) are executed with one (or more) DOF associated to each thread, while in the internal and external force calculation step (step 4) one element per thread is considered. While the code concerning each DOF and node is executed independently in steps 3 and 5, the assembly of elemental forces in step 4 requires the accumulation of values in the nodes, which is essentially a reduction process. Communication and synchronization between threads belonging to different blocks (and therefore using non-shared memory, see figure 1) in GPU is a difficult task, since CUDA supports native thread synchronization only at thread-block level. To avoid race conditions during forces accumulation, a *coloring* technique is employed: elements are divided into subsets such that all elements in a specific subset do not share any node. Nodal assembly of forces can thus be performed on one subset at a time, with different consecutive calls to the CUDA kernel, and threads are free to write their results without worrying about other threads accessing the same memory locations. This implementation already shows good performances with respect to the serial CPU implementation; but it is not fully exploiting the power of the GPU. The algorithm in the form presented in section 2.1 is not designed to be run on GPU and this limits the actual achievable speedup.

3.2. Enhanced GPU implementation

In the time integration scheme presented in section 2.1, the steps of the algorithm are basically loops, operating on different types of entities (DOFs, nodes, elements); this structure is exploited in the first GPU implementation described above, executing each step in separate CUDA kernels. The first objective of the enhanced implementation is to express the whole algorithm as a loop over a single type of entity. In this way, a single CUDA kernel is sufficient to perform all calculations concerning an entire time step, avoiding multiple kernel calls, with the relative overhead, and minimizing memory accesses.

Using DOFs as main entity would be convenient only in the two integration half-steps, while the other steps would be too difficult to be formulated on a per-DOF basis. A better approach is obtained looping over nodes, since at each node only the DOFs pertinent to that node are integrated at each time step. However, computation of internal and external forces, the most computationally expensive step, is hard to be performed on a per-node basis, since it requires integration over the elements. Each node should know which elements it belongs to, their properties and materials, and then force contribution coming from all those elements should be computed. This would cause intensive memory access, with wasted time to perform the same calculations more than once (considering also the lack of easy synchronization and communication between thread blocks).

All these problems can be solved by looping over the elements. Every element keeps all the information about its nodes in a local copy, not shared with other elements (see figure 2, where global versus local node numbering is shown). Consequently, each element performs time integration on its own nodes, and then it can proceed to forces calculation. All steps can thus be carried out in one single, long and articulated CUDA kernel.

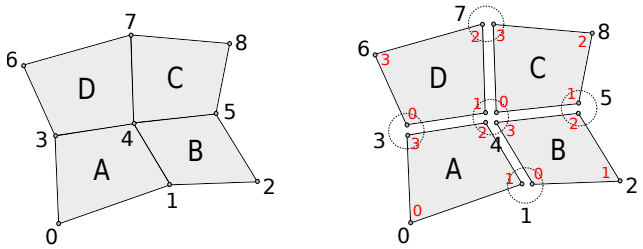


Figure 2: Example of mesh with four elements, indicated by the letters A, B, C, D. Global node numbering versus local node numbering: global node numbers are in black; each element has its own copy of the nodes, which are locally numbered in red.

Performing element by element computations, some operations need to be repeated several times: each element keeps its own copy of the local nodes and therefore has to build and integrate nodal quantities even if they are in common with other elements. Nevertheless, in this way only the force assembly step, where forces coming from neighboring elements are gathered, requires a communication between different elements. This approach is effective only in GPU implementations, where repeating the same calculation can be convenient if it means performing fewer memory accesses, since GPUs are much more powerful at executing floating point operations rather than accessing memory. Moreover, the amount of memory required to implement the GPU optimized central difference scheme is limited, thus some memory can be wasted by duplicating necessary information, if this leads to simplifications in the kernel structure or better memory access patterns. Finally, since computation of nodal forces is the most computationally intensive step, possible repetitions should concern other less demanding steps.

Another advantage of the proposed element-wise algorithmic structure is that when an element is considered, the only mesh connectivity information necessary to the CUDA kernel for each one of the element local nodes concerns the indexes of neighboring elements which share it and the corresponding local node index inside those elements. The algorithm is also independent of the element type: it could be applied, for example, to triangular elements (or other types of elements) without changing the code structure. The proposed GPU formulation can naturally cope with unstructured meshes. This is in contrast with many FEM solvers implemented in GPU, which usually require

structured grids.

3.2.1. Central difference method on GPU

In the standard time stepping algorithm with central difference method, the force calculation step is the only one that needs explicit communications: after the nodal forces are computed for each element separately, they need to be exchanged between neighboring elements, in order to compute the right nodal force contributions. A slight modification to the standard central difference scheme sketched in section 2.1 is introduced to avoid communications inside the same CUDA kernel. Following the algorithm of section 2.1 the acceleration update is conceptually considered the last operation of the time step. In the proposed approach, it becomes the first operation of the next time step, while the computation of nodal forces becomes the last one; i.e. for each element the following sequence of operations is carried out at each time step:

$$a_i^n = \frac{f_{E,i}^n - f_{I,i}^n - c_i\, v_i^{n-\frac{1}{2}}}{m_i + c_i\,(t^n - t^{n-\frac{1}{2}})}, \tag{3.1}$$

$$v_i^n = v_i^{n-\frac{1}{2}} + (t^n - t^{n-\frac{1}{2}})\, a_i^n \tag{3.2}$$

(computed only if required for energy check),

$$v_i^{n+\frac{1}{2}} = v_i^{n-\frac{1}{2}} + (t^{n+\frac{1}{2}} - t^{n-\frac{1}{2}})\, a_i^n, \tag{3.3}$$

$$u_i^{n+1} = u_i^n + \Delta t^{n+\frac{1}{2}}\, v^{n+\frac{1}{2}}, \tag{3.4}$$

$$\mathbf{F}_I^{n+1} = \text{internalForces}\left(\mathbf{U}^{n+1}\right), \tag{3.5}$$

$$\mathbf{F}_E^{n+1} = \text{externalForces}\left(\mathbf{U}^{n+1}\right). \tag{3.6}$$

In this case it is obviously necessary to perform an initialization step where initial forces, DOF values and half-step velocities are computed.

With this new solution scheme (3.1 - 3.6), there are two main advantages. Firstly, the accelerations a_i^n are calculated only in the step where they are used and do not need to be stored. Therefore, the only used global buffers are related to DOF values (nodal displacements and rotations), half-step velocities and forces. Furthermore, this procedure does not need synchronizations inside a time step. In the force calculation step each element computes only its contributions to the forces on the local nodes. Placing this step at the end of the kernel execution, we exploit the implicit synchronization occurring between successive kernel calls to ensure that, at the next time step, when the kernel is invoked again, each element can freely gather all the previous step force contributions to its local nodes coming from the neighboring elements, without worrying about data integrity and race conditions. The whole time step advancing procedure is thus contained in a single big CUDA kernel, with a schematic flow chart sketched in figure 3. The algorithm in pseudo-code is briefly described in algorithm 1.

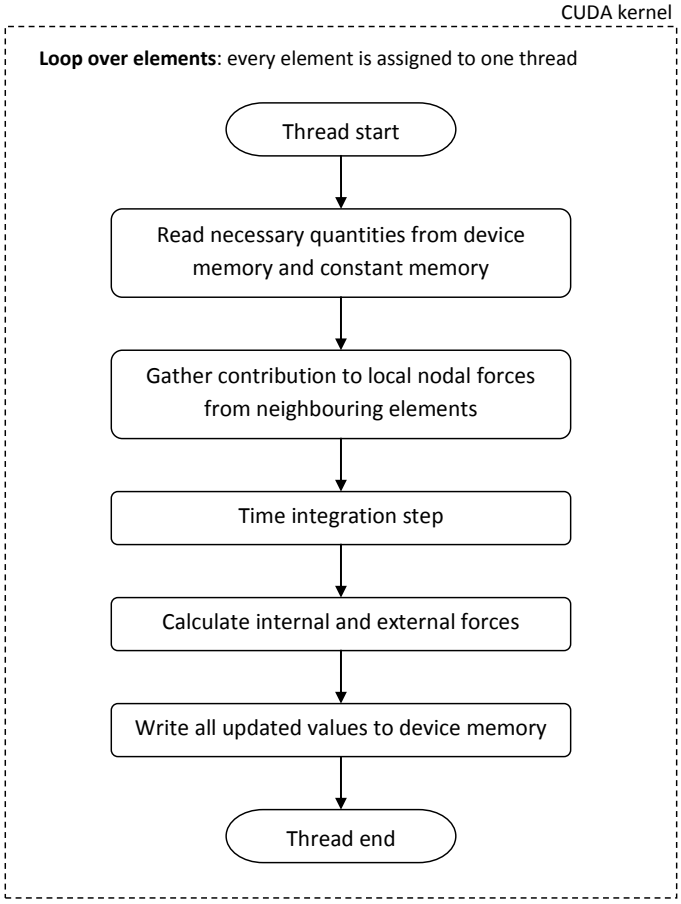


Figure 3: Steps of time integration kernel.

3.2.2. Memory management and performance improvement

Thanks to the structure described in the previous sections, the entire simulation data resides in GPU memory and transfers to or from host memory are not necessary (they are performed only to retrieve simulation results when needed or change boundary conditions or external loads). However, in order to maintain good performances, particular care must be taken in the organization and management of the data structures in device memory, such that non-optimal memory access patterns are minimized, in view of the relatively high cost of each individual memory access. Memory access optimization is achieved when memory access patterns suitable for coalescence are realized. Device memory is accessed using aligned memory transactions of 32, 64 or 128 bytes. When threads access the global memory it results in a certain amount of memory transactions, depending on the location and size accessed by each thread (figure 4); therefore, optimal throughput is achieved when memory accesses by the single threads can be coalesced into the smallest amount of transactions necessary to cover them. Devices supporting compute capability 2.x and higher are also able to cache memory transactions. Specifications about the inner handling of device memory accesses, together with the optimal

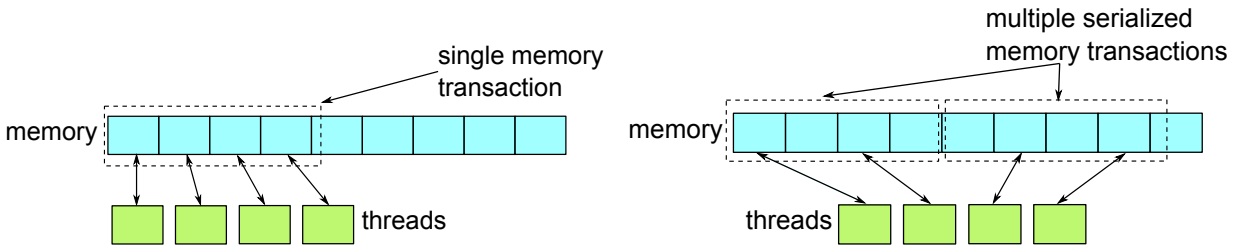
Algorithm 1 Time integration at time step n

for $i = 1, \ldots, NumberOfElements$ **do**
  read data of element i from previous time step
  for $k = 1, \ldots, NumberOfNeighbors[i]$ **do**
    collect forces from neighbor element k, accumulating them locally
  end for
  for $j = 1, \ldots, 4$ **do**
    compute acceleration $a^n_{i,j}$ using (3.1) on the local nodes j
    compute half-step velocity $v^{n+\frac{1}{2}}_{i,j}$ through (3.3)
    compute displacement $u^{n+1}_{i,j}$ using (3.4)
  end for
  evaluate forces, internal and external, on the element i
  write data of element i
end for

access patterns and other considerations are thoroughly addressed in [25].
Static data that do not change during a time step can be stored in constant memory, which ensures good access performance. Constant memory has however limited size, thus data concerning mesh structure have to be stored together with nodal DOF values in the much slower device memory. To preserve efficiency, they have therefore to be organized in optimal layouts to allow for coalesced access. Generally, depending on the specific situation, data of this kind can be organized in array-of-structures (AoS) or structure-of-arrays (SoA), each with its own advantages and disadvantages, to fully exploit the memory access coalescence and the memory cache. In some specific situations, even an hybrid approach can be effective (structure-of-arrays-of-structures, SoAoS). Without going into further details, in the presented implementation different approaches have been tested, but the best performances have been achieved organizing data in a SoA layout, which greatly helps in keeping memory read/write operations coalesced. Therefore, full coalescence is guaranteed in the whole kernel for all memory operations, with the only exception of internal force exchange: each element needs to gather all contributions to the internal forces from the neighboring elements, reading them according to the local connectivity arrays, thus without a good pattern of access. This problem can be slightly alleviated in the case of structured meshes by renumbering the nodes, but in general, with unstructured meshes, the accesses cannot be coalesced. A possible solution to this problem can be found exploiting the shared memory: threads in a block can read all the values needed by the entire block of threads into shared memory in a coalesced way and then from there each thread can access the values it needs. Nevertheless, for the present solver the performance impact of these non-coalesced read operations is generally negligible if compared to the computational time of one whole time step and the increased code complexity does not make it worth to be implemented.
To avoid read-and-write conflicts, a double-buffering approach is introduced for the storage of the element local forces. Following the solution scheme introduced in section 3.2.1, at the beginning of the kernel internal and external forces of the previous step are read for each element and at the end the new ones are computed and stored. To avoid synchronization, input and output forces are respectively read and written using different buffers, which are then swapped when executing the following time step.
As described above, with the proposed algorithm calculations are performed on a per-element basis, with all elements keeping their own copy of the local nodes. If compared to a standard implementation, this means that some nodal quantities are duplicated. It is possible to estimate the memory consumption of this algorithm in order to prove that wasting memory on duplicating these information is not an issue. As far as GPU memory is concerned (usually GPU boards are equipped with less memory than the amount of available RAM, thus the bottleneck is GPU memory size), the following buffers are allocated:

- Read/write buffers:
  - DOF displacement and rotation values: 20 x (number of elements) reals;
  - DOF half-step velocities: 20 x (number of elements) reals;
  - nodal directors: 4 x (number of elements) x (3 components) reals;
  - nodal basis: 2 x 4 x (number of elements) x (3 components) reals;
  - element local forces: 20 x (number of elements) x 2 (because of the double-buffering) reals.
- Read-only buffers:
  - initial nodal directors: 4 x (number of elements) x (3 components) reals;
  - mass on the DOFs: 20 x (number of elements) reals;
  - element material properties and external pressures: 4 x (number of elements) reals;
  - external nodal forces: 4 x (number of elements) x (3 components) reals;
  - initial node coordinates: 4 x (number of elements) x (3 components) reals;
  - neighbors array: 2 x 4 x (max. number of neighbors) x (number of elements) integers.

Let us assume that 6 is the maximum number of elements sharing the same node. Summing up all contributions, it results that each element requires 176 reals and 48 integers. Considering double-precision arithmetic, so that reals are 8



(a) Memory accesses coalesced into one single transaction.

(b) Non fully coalesced memory accesses: two transactions are required.

Figure 4: Example of coalescence of memory accesses.

bytes long, and 32-bit integers, each element requires 1600 bytes of storage. A low-range gaming GPU nowadays has often at least 1 GB of video RAM. NVIDIA *Tesla* computing boards are equipped usually with several GBs of memory. For each GB, theoretically 671088 elements can be stored. This consideration ensures that duplicating entities at element level is not an issue in the present approach, since that with few GBs of memory real-life engineering problems can be solved. Furthermore, if the memory is not enough to contain the data of a very large discretization, the structure of the presented solver allows the mesh to be partitioned and asynchronously streamed to GPU memory while executing, without substantial changes in the code (just some specialized management of elements at sub-domain interfaces).

Finally, additional performance improvements have been achieved by means of specific implementation expedients. For example, the code needs to know whether a DOF is constrained or not. Instead of storing this information in a separate buffer, it is contained in the sign of the mass: if it is negative, it means that the corresponding DOF is constrained. Even if it introduces a branch in the code, this little optimization is fundamental to avoid an additional memory operation and results in an actual decrease of computational time.

From earlier tests with CUDA profiling tools, it emerged that in the presented CUDA kernel the biggest cause of performance drop is caused by register spilling: there were too many local variables to fit the available registers, thus some of them were being stored in local memory (which has the same latency as global memory) causing a significant drop in performance. This problem was alleviated in two steps. In order to avoid storing values in temporary variables for the whole time step process, read and write operations were moved before and after the actual code using them, and not at beginning or end of the kernel as it is usually done. Furthermore, in order to help the CUDA optimizer, a meticulous process of local variable scope tuning (enforced dividing manually the source code in blocks using curly brackets) was performed, forcing variable reuse whenever possible, with good results in terms of performance gain (around 20%). Nevertheless, it is important to highlight that, although alleviated, register spilling remains a main concern. This is however related to the choice of using a single long CUDA kernel: tests have shown that this single-kernel structure remains the best approach, when compared to implementations that split calculations over different kernels, even if it reaches a sub-optimal level of occupancy. In addition, the proposed implementation supports different thread block sizes. At the beginning of a simulation, a simple and quick auto-tuning step is performed: different block sizes are tried and the best one is kept for the entire simulation. This ensures also a level of automatic adaptivity to different hardware.

4. Numerical examples

In order to check the solver accuracy and to compare performances, in this section three different test problems are presented. The performances achieved by the proposed GPU solver in the first two test cases are analyzed and compared against the serial CPU implementation of Abaqus, in order to have a comparison with a widely known commercial software. In this case, Abaqus performance has to be regarded just as a reference, since Abaqus Explicit is highly optimized for multicore CPU execution (64/128/256 cores), a standard installation for many commercial customers and not for serial, single CPU implementations of the type here considered. Abaqus simulations are executed on a single core of an AMD *Phenom X4 9950 Black Edition* CPU at 2.6 GHz. GPU simulations run on the same machine, equipped with a NVIDIA *GeForce GTX 470* board, provided with 448 CUDA cores (14 SM, 32 cores/SM). The simulations of the first two examples are performed within a Linux environment, without graphical user interface (to not interfere in any way with GPU computations). Considering the third test case, the GPU solver is executed with increasingly refined meshes, until the GPU memory is almost completely filled. For this problem, a more powerful GPU board is used: a *Tesla K20* board, equipped with 2496 CUDA cores and 4800 MB of memory, on a different host machine running Microsoft Windows. The fact that the code has been tested in different configurations shows how the proposed implementation does not depend on the hardware or the operating system.

The performance studies are conducted simulating the large displacement deformation characteristics of the given problem with different meshes. All meshes have been created in Abaqus. Specifically, classical *S4* four node shell elements, without reduced integration, have been used for the analyses with Abaqus, while the MITC4 four node shell elements described at the beginning of Section 2 have been used for the analyses with the GPU code. For each mesh, the computational time taken to perform a complete simulation by Abaqus and by the presented solver is recorded. These timings do not include any pre-processing or post-processing, but only the actual computational time. It is important to highlight that the duration of pre-processing operations needed by the GPU solver is negligible (they substantially consist in building connectivity arrays, necessary to handle the unstructured mesh, and the kernel thread block size auto-tuning process, which is optional). In order to guarantee a fair comparison, the computational time taken by Abaqus is measured as the wall-clock time (i.e. the time measured by the computer clock, outside the application) taken by the explicit solver executable to complete the simulation. The amount of I/O to file performed by both solvers is kept as similar as possible and in general minimized.

4.1. Uniformly loaded circular plate

A circular plate of an isotropic homogeneous material is analyzed under constant external pressure and simply supported edge conditions, i.e. out-of-plane displacements at the plate edge are constrained, while free rotations and in-plane displacements are allowed. The material is characterized by a Young modulus E of 206000 MPa and a Poisson ratio ν of 0.3. The plate radius and thickness measure 10 mm and 0.1 mm, respectively. The external applied pressure is of 0.0001 MPa. Considering a static analysis of the problem, under small displacements assumption and Reissner-Mindlin shell theory, the nodal displacements can be computed analytically for verification purposes [26]. Figure 5 shows an example of the deformed mesh. The results obtained with Abaqus and with the GPU code, in terms of the vertical displacement of the plate central point versus time, for the coarsest mesh considered, are shown in figure 6.

For each mesh the total time taken to perform a complete simulation of 120 seconds is recorded. Results obtained in terms of computational times with double-precision and single-precision floating point arithmetic, using seven meshes of increasing finesse, are summarized in tables 1 and 2. Abaqus is equipped with a time step size estimation algorithm that adapts it during the simulation, trying to keep it as large as possible, while preserving stability. For the sake of comparison, the presented GPU solver uses the smallest (to guarantee stability) time step size calculated by Abaqus throughout the simulation and keeps it fixed. The columns marked as *Abaqus adaptive* show timings of Abaqus with time step adaptation turned on, while columns marked as *Abaqus fixed* present timings with fixed time step size equal to the one used by the GPU solver.

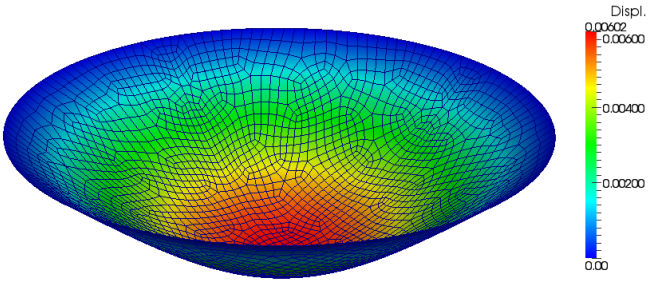


Figure 5: Circular plate. Deformed mesh after 100 seconds (3023 elements). Displacements amplified by scaling factor 1000.

Table 1: Circular plate. Computational time in seconds. Double-precision arithmetic.

Elem.	Abaqus adapt.	Abaqus fixed	GPU impl.
500	24.86	32.13	2.46
1125	88.02	118.54	5.29
2000	216.56	291.86	9.38
3125	435.28	576.10	15.24
4500	769.04	1023.15	24.46
8000	1884.61	2472.49	59.76
12,500	3753.52	4845.03	111.51

Table 2: Circular plate. Computational time in seconds. Single-precision arithmetic.

Elem.	Abaqus adapt.	Abaqus fixed	GPU impl.
500	13.57	18.32	1.18
1125	46.82	63.77	2.37
2000	115.92	158.13	4.05
3125	231.33	312.44	6.36
4500	408.17	541.74	9.73
8000	988.34	1318.71	24.12
12,500	1973.87	2606.35	42.43

Considering the mesh with the largest number of elements, in double-precision, the GPU solver performs the same simulation more than 33 times faster than Abaqus, when time step adaptation is turned on. If a comparison is made fixing the time step for both solvers, the gained speedup is even higher, over 43x (see Fig. 7). While it is not always fair or meaningful to do this kind of comparisons, it is indeed useful in order to have an estimation of the performance obtainable relatively to a common and widely known software. Moreover, it is important to highlight that these simulations are performed using a common GPU board suited for video-gaming and not specifically conceived for scientific computing. Despite the good results shown using double-precision arithmetic, these kind of video boards are much more suited for 3D graphics and single-precision. In fact, running

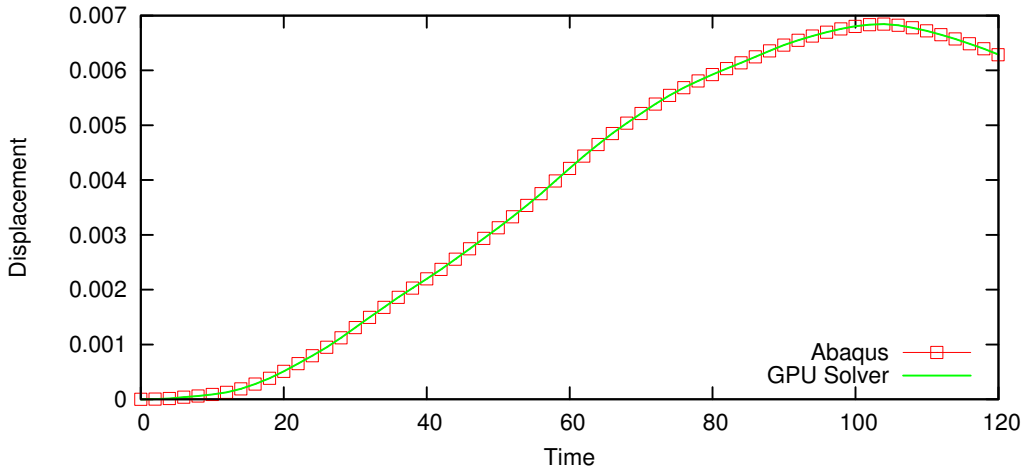


Figure 6: Circular plate. Comparison of the time evolution of the central point's vertical displacement.

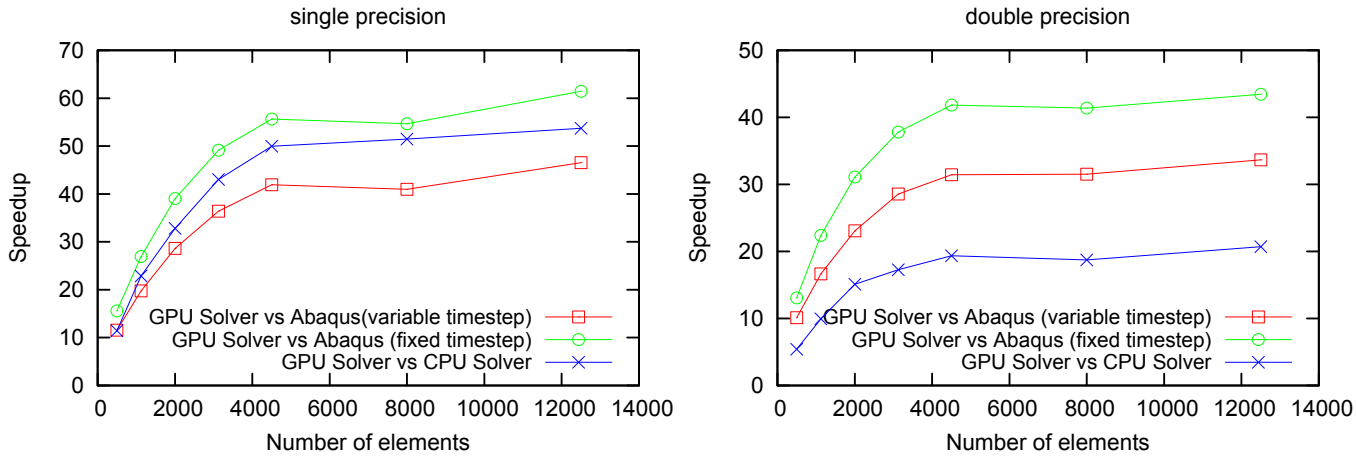
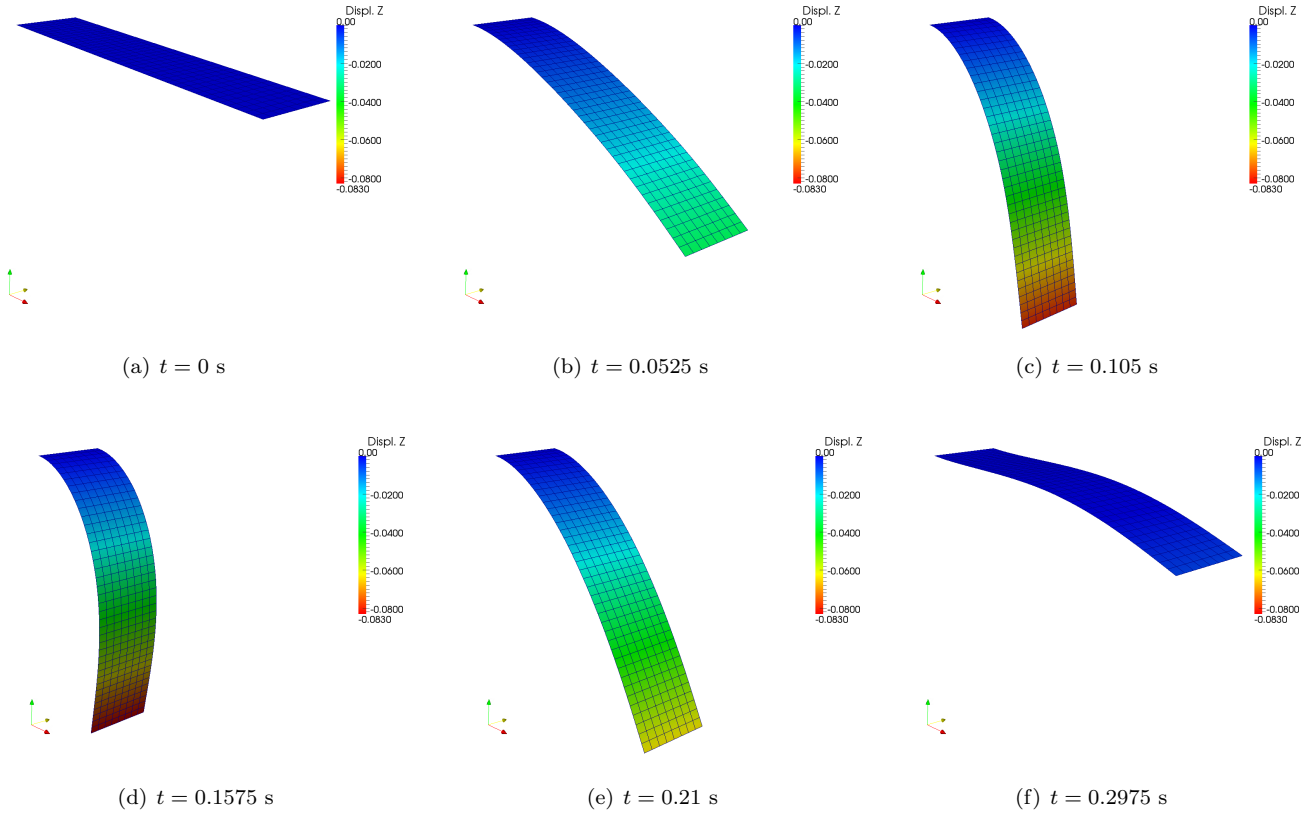


Figure 7: Circular plate. Speedup for single-precision and double-precision arithmetic.

the simulations with single-precision arithmetic, speedups of 46x and 61x are obtained considering Abaqus adaptive and fixed time step configurations respectively. Using computationally-oriented GPU boards even better performances are achievable and the speedup curve would saturate at a higher value (see Fig. 7). The same problem has also been tested using the first GPU implementation (see section 3.1) obtaining a computing time at least 5 times larger with respect to the results of tables 1-2.
It is also interesting to notice that, even with the finest considered mesh, the GPU implementation is able to compute a simulation of 120 seconds in less than 112 seconds of computational time. This means that the problem can be simulated completely in real-time, using the full MITC4 shell elements and without requiring reduced order approaches.

4.2. Clamped rectangular plate

A rectangular plate is fixed at one edge and subjected to a tip load. The load (0.00156 MPa) is applied at time $t = 0$ and then kept constant during the whole simulation. The plate is 100 mm long, 20 mm wide and 0.2 mm thick. It is made of a flexible material characterized by Young modulus E equal to 1768 MPa, Poisson ratio ν equal to 0.3 and a density ρ of 3000 kg/m^3. A dynamic analysis of the plate is carried out for 0.35 seconds of simulation. This mesh is a good test for the solver for two reasons: it involves large displacements, so the accuracy of the small strains – large displacements formulation is tested, and the geometry can be discretized in a structured way: this is well suited for the GPU implementation because the regularity of the grid helps the internal forces intercommunication process (even if the code is conceived for unstructured grids). In figure 8 the deformed mesh at different time instants obtained with the GPU solver is shown.The evolution in time of the vertical displacement of the cantilever tip, obtained by Abaqus and the GPU code, for the coarsest mesh considered, is shown in figure 9.
In order to compare performances of the GPU implementation against Abaqus, different mesh refinements are tested.



(a) $t = 0$ s (b) $t = 0.0525$ s (c) $t = 0.105$ s

(d) $t = 0.1575$ s (e) $t = 0.21$ s (f) $t = 0.2975$ s

Figure 8: Deformations of the cantilever at different times (no displacement scaling factor used).

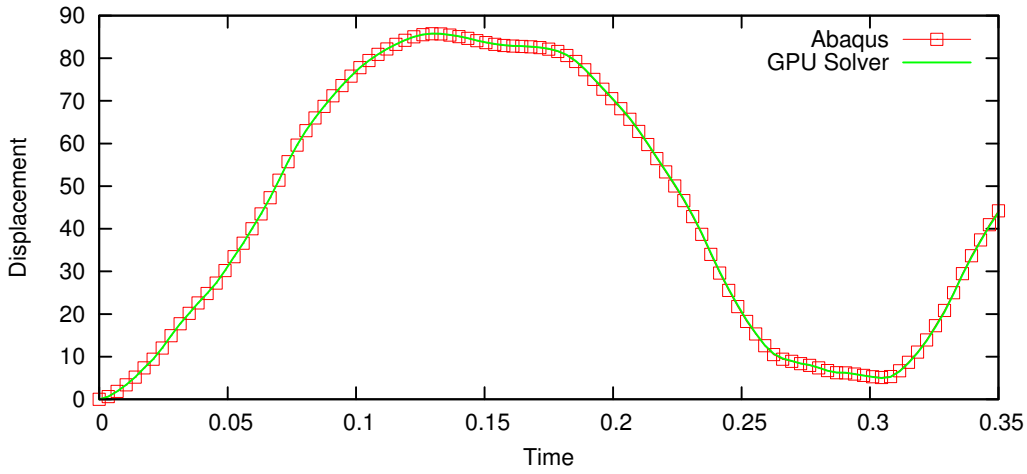


Figure 9: Clamped rectangular plate. Comparison of the time evolution of the plate tip's vertical displacement.

Total computational times taken by Abaqus (with both adaptive and fixed time steps) and the GPU implementation are summarized in tables 3 and 4 for double and single-precision arithmetic respectively. Thanks to the time step adaptation process, Abaqus calculates and uses during the whole simulation only two different time step sizes: a first smaller one, used only for the first time increment, and a second one used in the rest of the simulation. In this situation, however, these two time step sizes are not so different, causing an unusual consequence: Abaqus runs faster using a fixed time step (the smaller one) with no time step adaptivity. This means that, in this case, the time step size calculation process takes more time than what it manages to save using bigger time increments. For the sake of comparison, the GPU solver uses, as always, the smallest time step size calculated by Abaqus. It is

important to underline that, with respect to the previous test case, this problem requires a much smaller time step to describe the large displacements of the structure. Considering the finest mesh, for this example the time step is of the order of 10^{-7}, while for the previous one it was of the order of 10^{-3}; therefore the simulation cannot be performed in real-time with the employed GPU board.

Table 3: Clamped rectangular plate. Computational time in seconds. Double-precision arithmetic.

Elem.	Abaqus adapt.	Abaqus fixed	GPU impl.
80	57.79	57.99	16.61
160	180.64	179.10	29.87
320	374.93	369.28	38.61
640	1294.34	1277.35	77.44
1280	2898.43	2858.18	113.54
2560	10214.82	10094.71	272.56
5120	22891.90	22517.00	499.73
10,240	81117.21	79891.72	1755.99

Table 4: Clamped rectangular plate. Computational time in seconds. Single-precision arithmetic.

Elem.	Abaqus adapt.	Abaqus fixed	GPU impl.
80	38.65	39.23	6.80
160	110.58	108.14	12.26
320	216.55	217.19	18.13
640	709.78	708.25	37.64
1280	1541.46	1553.88	47.34
2560	5541.72	5504.04	112.27
5120	12297.11	12316.41	194.08
10,240	43813.10	43300.00	685.66

In this simulation, the efficiency achieved by the GPU implementation is even higher than the one obtained in the uniformly loaded circular plate test case: speedup is over 46x in double-precision and increases to 63x in single-precision. This is probably due to the fact that the regular geometry and discretization permit to take advantage of the whole potential of the GPU implementation, exploiting the performance of the parallelized algorithm. This is clear also by looking at the speedup graphs in figure 10: the software seems to reach a saturation point, where capabilities of the GPU are used at their maximum and the speedup cannot increase any further. Obviously, these saturation curves depend heavily on the GPU hardware: with computationally-oriented GPU boards they are expected to reach higher peak values.

4.3. Pinched cylinder

To assess the solver performance in a more severe application, one of the numerical tests discussed in [27] is considered. A cylinder clamped to one end is analyzed under the load of two opposite forces. The setup is the same as in [27]: the cylinder has radius equal to 1.016 m, length of 3.048 m and thickness of 0.03 m; the material considered has Young modulus of $2.0685 \cdot 10^7$ MPa, Poisson coefficient of 0.3 and density of 10^3 Kg/m^3. The two opposite concentrated loads have magnitude of 500 MPa. Figure 11

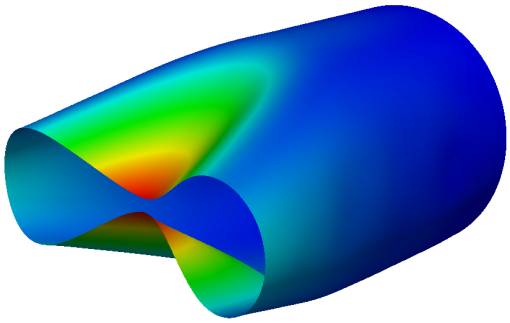


Figure 11: Pinched cylinder. Deformed configuration (contour plot shows displacement magnitude).

Table 5: Pinched cylinder: performances of GPU implementation for different meshes.

Elements	DOFs	Memory (MB)	Time(s)	Speed
294912	1478400	423	1259.67	62.21
524288	2626560	752	2242.23	34.95
819200	4102400	1175	3505.95	22.35
1280000	6408000	1836	5471.68	14.32
1843200	9225600	2644	7879.08	9.95
2508800	12555200	3598	10719.60	7.31
3276800	16396800	4700	14277.20	5.49

shows a snapshot of the deformed structure and Figure 12 compares the load-displacement curve with the results of [27].

With this kind of geometry it is simple to refine the mesh and thus analyze the solver performance with a number of degrees of freedom regularly increasing. For all meshes, a time step size $\Delta t = 1.276110^{-5}$ seconds is considered, so that the simulation is always stable. For each mesh the simulation is run for 78363 time steps, which correspond to about 1 second of physical time. With respect to the previous examples, this performance test is conducted using a more powerful GPU board: a NVIDIA *Tesla K20* (2496 CUDA cores) equipped with 4800 MB of memory. To make results easier to compare, the GPU thread-block size auto-tuning feature is turned off: all blocks are fixed to 32×32 threads. Information about the considered meshes together with the total computational times are summarized in table 5.

The column *Speed* lists the number of time steps processed per second, which multiplied by the time step size gives the physical time span which can be simulated in one second of computation. The performance of the proposed implementation can be analyzed in figure 13, where the computational time is reported over the total number of degrees of freedom. The relation is almost perfectly linear: this means that the proposed implementation has not reached a saturation point of the computational power, even using the finest mesh considered. Furthermore, with this analysis we pushed the number of elements of the mesh to the limit given by the amount of device memory on the

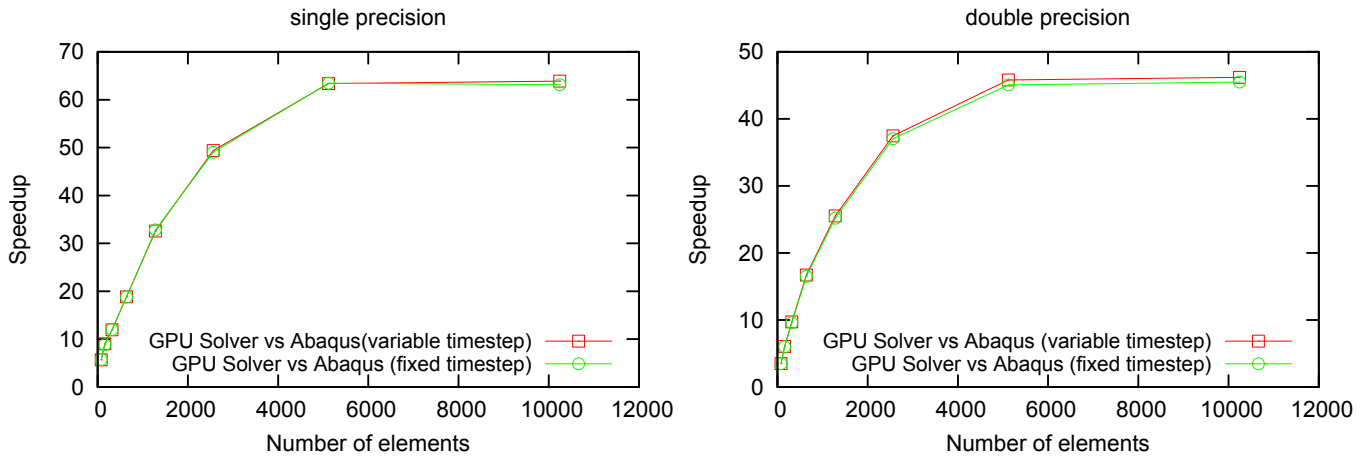


Figure 10: Clamped rectangular plate. Speedup for single-precision and double-precision arithmetic.

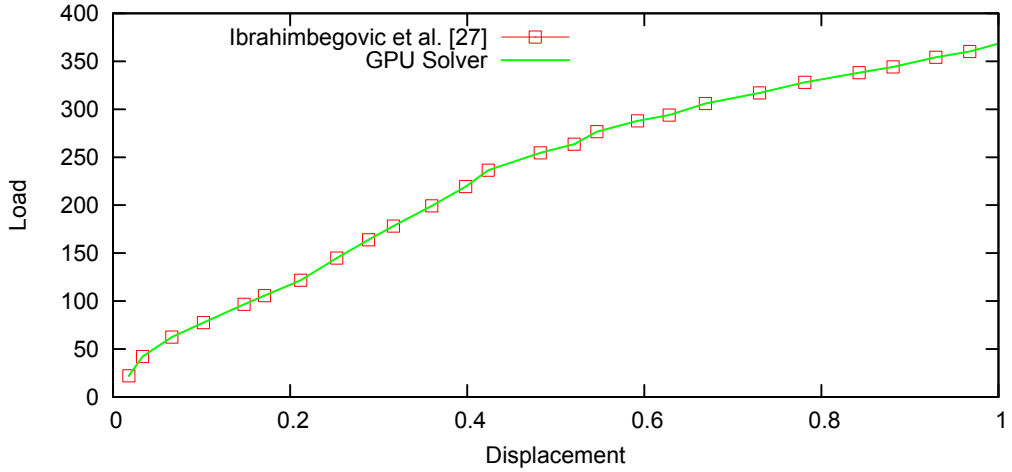


Figure 12: Pinched cylinder. Load-displacement curve: comparison against [27].

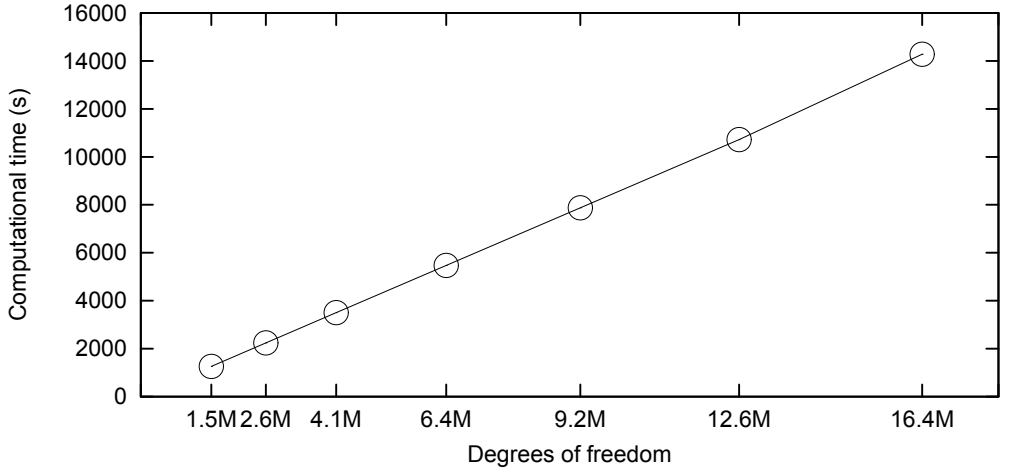


Figure 13: Pinched cylinder: total computational time vs number of DOFs.

GPU board. In table 5, in the column *Memory*, theoretical memory consumption estimates are reported (they differ from actual consumption measures by few megabytes). Therefore, the trend shown in figure 13 is not affected even

though the last mesh considered fills almost completely the available GPU memory.

5. Conclusions and future developments

In this work a double-precision, explicit dynamics nonlinear finite element shell solver on unstructured grids, designed specifically to carry out the whole computation on a GPU has been introduced. To fully exploit the computational power of GPUs, an innovative implementation of the central difference scheme is proposed. Results show how these kinds of problems are well suited for GPU implementation and thanks to the new algorithmic formulation and the careful memory management, the GPU-only implementation is able to reach very high performances with limited memory consumption.
Explicit time integration requires a large amount of very small time steps to be performed, but on GPU they can be executed extremely quickly and overall performances are satisfactory. It is also important to emphasize that during these simulations the CPU is practically idle. This fact can be exploited in more complex situations where different systems have to be simulated concurrently: keeping computation on GPU frees the CPU to perform other demanding tasks.
The GPU solver has been built from scratch in order to be completely free from constraints of an existing design and to be organized and prepared for execution on GPU hardware from the beginning of the development. It is often possible to directly port a CPU application to GPU with some adjustments; but it is only designing the algorithms from scratch, with the clear purpose of running on GPU in mind, that the best performances can be achieved.
Last but not least, with GPUs high performance computing can be obtained with very low power consumption. The GPU board should approximately need 225 W while dealing with real applications. Obtaining the same performances on a CPU cluster would definitely require much more power.
This work constitutes a starting point for the future solution of more complex problems. First of all, the GPU implementation should be tested on computationally-oriented GPU boards to assess the achievable performances in more demanding real-life engineering case problems, possibly including various sources of material nonlinearity. Another important topic of future developments will be the introduction of contact algorithms in the GPU implementation and the possibility of simulating fracture propagation. With all these ingredients, the code will be able to guarantee effective computing times (near real-time in some special cases) for the simulation of engineering problems of real use.

6. Acknowledgements

The authors gratefully acknowledge the support of NVIDIA Corporation with the donation of the *Tesla K20* GPU used for this research. This work has been partially supported by Regione Lombardia and CINECA Consortium through a LISA Initiative (Laboratory for Interdisciplinary Advanced Simulation).

References

[1] S. Georgescu, P. Chow, H. Okuda, GPU Acceleration for FEM-based structural analysis, Archives of Computational Methods in Engineering 20 (2) (2013) 111–121. doi:10.1007/s11831-013-9082-8.
[2] A. Kl ockner, T. Warburton, J. Bridge, J. S. Hesthaven, Nodal discontinuous Galerkin methods on graphics processors, Journal of Computational Physics 228 (21) (2009) 7863–7882. doi:10.1016/j.jcp.2009.06.041.
[3] H. Zhou, G. Mo, F. Wu, J. Zhao, M. Rui, K. Cen, GPU implementation of lattice Boltzmann method for flows with curved boundaries, Computer Methods in Applied Mechanics and Engineering 225–228 (2012) 65–73. doi:10.1016/j.cma.2012.03.011.
[4] F. Kuznik, C. Obrecht, G. Rusaouen, J. J. Roux, LBM based flow simulation using GPU computing processor, Computers & Mathematics with Applications 59 (7) (2010) 2380–2392. doi:10.1016/j.camwa.2009.08.052.
[5] A. Sunarso, T. Tsuji, S. Chono, GPU-accelerated molecular dynamics simulation for study of liquid crystalline flows, Journal of Computational Physics 229 (15) (2010) 5486–5497. doi:10.1016/j.jcp.2010.03.047.
[6] J. Zheng, X. An, M. Huang, GPU-based parallel algorithm for particle contact detection and its application in self-compacting concrete flow simulations, Computers & Structures 112113 (0) (2012) 193 – 204. doi:dx.doi.org/10.1016/j.compstruc.2012.08.003.
[7] L. Iuspa, P. Fusco, E. Ruocco, An improved GPU-oriented algorithm for elastostatic analysis with boundary element method, Computers & Structures 146 (0) (2015) 105 – 116. doi:dx.doi.org/10.1016/j.compstruc.2014.08.009.
[8] M. Papadrakakis, G. Stavroulakis, A. Karatarakis, A new era in scientific computing: Domain decomposition methods in hybrid CPU–GPU architectures, Computer Methods in Applied Mechanics and Engineering 200 (13–16) (2011) 1490–1508.
[9] G. R. Joldes, A. Wittek, K. Miller, Real-time nonlinear finite element computations on GPU-Application to neurosurgical simulation, Computer Methods in Applied Mechanics and Engineering 199 (2010) 3305–3314. doi:10.1016/j.cma.2010.06.037.
[10] R. Mafi, S. Sirouspour, GPU-based acceleration of computations in nonlinear finite element deformation analysis, International Journal for Numerical Methods in Biomedical Engineeringdoi:10.1002/cnm.2607.
[11] Y. Cai, G. Li, H. Wang, G. Zheng, S. Lin, Development of parallel explicit finite element sheet forming simulation system based on GPU architecture, Advances in Engineering Software 45 (1) (2012) 370–379. doi:10.1016/j.advengsoft.2011.10.014.
[12] A. Karatarakis, P. Karakitsios, M. Papadrakakis, GPU accelerated computation of the isogeometric analysis stiffness matrix, Computer Methods in Applied Mechanics and Engineering 269 (2014) 334–355. doi:10.1016/j.cma.2013.11.008.
[13] B. Mihaila, M. Knezevic, A. Cardenas, Three orders of magnitude improved efficiency with high-performance spectral crystal plasticity on GPU platforms, Int. J. Numer. Meth. Engng (2014) n/adoi:10.1002/nme.4592.
[14] F. Fritzen, M. Hodapp, M. Leuschner, GPU accelerated computational homogenization based on a variational approach in a reduced basis framework, Computer Methods in Applied Mechanics and Engineeringdoi:10.1016/j.cma.2014.05.006.
[15] R. Grimes, R. Lucas, G. Wagenbreth, Progress on GPU implementation for LS-DYNA implicit mechanics, in: 8th European LS-DYNA Users Conference, Strasbourg, 2011.
[16] S. Tomov, J. Dongarra, M. Baboulin, Towards dense linear algebra for hybrid GPU accelerated manycore

systems, Parallel Computing 36 (5–6) (2010) 232–240. doi:10.1016/j.parco.2009.12.005.
[17] O. Schenk, M. Christen, H. Burkhart, Algorithmic performance studies on graphics processing units, Journal of Parallel and Distributed Computing 68 (10) (2008) 1360–1369. doi:10.1016/j.jpdc.2008.05.008.
[18] A. Cevahir, A. Nukada, S. Matsuoka, High performance conjugate gradient solver on multi-GPU clusters using hypergraph partitioning, Computer Science - Research and Development 25 (1–2) (2010) 83–91. doi:10.1007/s00450-010-0112-6.
[19] G. Sharma, A. Agarwala, B. Bhattacharya, A fast parallel Gauss Jordan algorithm for matrix inversion using CUDA, Computers & Structures 128 (0) (2013) 31 – 37. doi:10.1016/j.compstruc.2013.06.015.
[20] D. Chapelle, K. J. Bathe, The finite element analysis of shells - Foundamentals, Springer, 2003.
[21] K. J. Bathe, Finite element procedures, Prentice-Hall, 1996.
[22] K. J. Bathe, E. Dvorkin, A formulation of general shell elements - the use of mixed interpolation of tensorial components, International Journal for Numerical Methods in Engineering 22 (1986) 697–722.
[23] T. Belytschko, W. K. Liu, B. Moran, Nonlinear Finite Elements for Continua and Structures, Wiley, 2000.
[24] T. Belytschko, J. I. Lin, C. S. Tsay, Explicit algorithms for the nonlinear dynamics of shells, Computer Methods in Applied Mechanics and Engineering 42 (2) (1984) 225–251.
[25] NVIDIA, Corporation, CUDA programming guide (2013). URL `https://developer.nvidia.com/cuda-zone`
[26] S. Timoshenko, S. Woinowsky-Krieger, Theory of plates and shells, Vol. 2, McGraw-hill New York, 1959.
[27] A. Ibrahimbegovic, B. Brank, P. Courtois, Stress resultant geometrically exact form of classical shell model and vector-like parameterization of constrained finite rotations, Int. J. Numer. Meth. Engng. 52 (11) (2001) 1235–1252. doi:10.1002/nme.247.