DEM-based simulation of concrete structures on GPU

Marie Durand, Philippe Maurice Marin, François Faure, Bruno Raffin

DEM based simulation of concrete structures on GPU

Marie Durand[*] — Philippe Marin[**] — François Faure [***] — Bruno Raffin [*]

[*] *INRIA Rhône-Alpes* [**] *3S-R (UJF/INPG/CNRS)* [***] *LJK (UJF/INPG/UPMF/CNRS/INRIA)*

ABSTRACT. *The benefit of using the Discrete Element Method (DEM) for simulations of fracture in heterogeneous media has been widely highlighted. However modeling large structure leads to prohibitive computations times. We propose to take advantage of Graphics Processor Units (GPUs) to reduce the computation time, taking advantage of the highly data parallel nature of DEM computations. GPUs are massively parallel coprocessors increasingly popular to accelerate numerical simulations. We detail our algorithm and implementation of the discrete element method (DEM) on GPU and present performance results for simulations of rock impact on a concrete slab, before to discuss the pro and cons of moving such computation to the GPU.*

RÉSUMÉ. *Pour simuler des structures soumises à de la fracturation, la méthode des éléments discrets (DEM) constitue un outil très efficace. Cependant la modélisation de grandes structures est très coûteuse en opération. Suite à l'observation que ces calculs sont fortement data-parallèles, nous proposons de tirer partie des processeurs graphiques (GPUs) pour réduire le temps de calcul. Les GPUs sont des coprocesseurs massivement parallèles de plus en plus utilisés pour accélérer des simulations numériques. L'algorithme et l'implantation sur le GPU sont détaillés puis nous présentons les résultats obtenus pour une simulation d'impact sur dalle en béton.*

KEYWORDS: *GPU, DEM, reinforced concrete structures*

MOTS-CLÉS : *GPU, DEM, structures en béton armé*

1re soumission à *EJECE*, le June 21, 2012

1. Introduction

The design of some particular civil engineering structures must take into account the risk of severe dynamic loadings due to natural or anthropogenic hazards such as rock falls, aircraft or missile impacts. Often, these severe loadings lead to localized fractures and fragmentation in the concrete structure. The Discrete Element Method (DEM) (Cundall *et al.*, 1979) is an appropriated method for modeling such discontinuities. The model uses disordered assembly of spherical and rigid elements of different sizes and masses to reproduce an isotropic and homogeneous behavior at a macroscopic scale. This method is very well adapted to dynamical problems, and does not rely upon any assumption about where and how a crack or several cracks occur and propagate as the medium is naturally discontinuous.

These DEMs were used first to model the behavior of granular materials, but they also provide very accurate results for cohesive materials like concrete (D'addeta *et al.*, 2002). The studies of Camborde in 2D (Camborde *et al.*, 2000) or Rousseau in 3D (Rousseau *et al.*, 2008) demonstrated the efficiency of such a discrete approach to deal with impact problems on reinforced concrete structures. They also pointed out the heavy computational load of DEM, limiting its use to small structures. To reduce the computational cost, we can use a coupling between the Discrete Element Method and the Finite Element (FE) Method (Xiao *et al.*, 2004), (Dhia *et al.*, 2005), (Frangin *et al.*, 2006), (Rousseau *et al.*, 2009). In the vicinity of the impact, where important non-linear phenomena occur, the medium will be modelled by means of Discrete Elements (DEs). The use of the FE method far from the impacted area is a way to reduce this limitation since in most cases severe degradation phenomena are localized in the vicinity of the impact.

But for shell structure (Rousseau *et al.*, 2010) impacted by large projectile like an aircraft, the size of the area that is represented by DEs is very important leading to an important computation time. Code parallelization is a classical approach to decrease simulation time or enable larger simulations. The goal of this paper is to evaluate the benefits of the parallelization of DEM based concrete structure simulation on Graphics Processing Unit (GPU). GPUs are coprocessors, usually having their own memory, communicating with the CPU through the PCI Express bus. GPUs were first dedicated to 3D graphics rendering, but as GPUs evolved towards more programmable architectures capable of executing user developed codes, it became possible to use them for performing generic computations. Today, using GPUs as general coprocessors is a major trend in high performance computing, usually called General Purpose GPU (GPGPU). Their high peak performance associated with their moderate cost make them good candidates to dramatically boost the computing power of a PC or the node of a super computer. Some GPUs are dedicated to GPGPU like the NVIDIA Tesla family (660 GigaFlops peak double precision floating point performance for the NVIDIA Tesla M2090), and software environments like OpenCL or CUDA are targeted at GPGPU programming. GPUs are highly parallel architectures, significantly diverging from CPUs (*cf.* section 3). An implementation on GPU can

lead to a significant performance improvement over a CPU implementation (Lee *et al.*, 2010).

However, such performance gains often require a significant programming effort and may be limited if the parallelism the application can exhibit does not match the GPU architecture. DEMs are good candidates for an efficient GPU implementation. A significant part of DEM related computations are data parallel (one instruction can be executed in parallel on many different data) and memory needs of today's DEM simulations usually fit the GPUs capabilities.

We first give a brief discription of the DEM we use in Section 2. We then sketch in Section 3 the principles of GPU architecture and programming, and their consequences on DE simulation. Our implementation is presented in Section 4, and we present experimental results in Section 5, showing speed-ups of an order of magnitude compared to an execution on a single core CPU. We finally conclude and discuss future work in Section 6.

2. Discrete Element Model

The DEM is based on the modeling of the continuum by means of rigid particles with 6 degrees of fredoom, 3 translations and 3 rotations. Interaction laws between DEs determine the macroscopic constitutive behavior. In the early developments particle interactions relied on friction laws for non-cohesive materials like sands (Cundall *et al.*, 1979). Interaction laws for cohesive materials were defined later on (Hentz *et al.*, 2004). To guarantee reasonable calculation durations, a model based on the Distinct Elements Model (Cundall *et al.*, 1979) with rigid spheres was chosen. Two types of interaction are defined. The initial interaction between two elements is generally a link interaction (the two elements are not necessarily in contact). Initially, two DEs interact if the distance between their centroids is less than a given radius of interaction. During the simulation additional interactions of contact type can be added. For concrete material, we used a modified Mohr-Coulomb model with softening (Rousseau *et al.*, 2008). More sophisticated laws taking into account compaction phenomena can be elaborated (Tran *et al.*, 2011) but they are not necessary to model a thin slab mainly subject to flexion and tension effects.

We can find in (Rousseau *et al.*, 2008) a procedure to identify all the material parameters based on the simulation of quasi-static compression and tension tests. The point is to identify local parameters to model macroscopic values such as compressive strength σ_c, tensile strength σ_t and fracture energy G_f. Special links are used to represent steel reinforcements and steel-concrete interface (Potapov *et al.*, 2012). In this paper, we rely on the interaction laws proposed by (Rousseau *et al.*, 2008) and (Potapov *et al.*, 2012). Refer to these publications for more details.

3. GPU Architecture and DE Parallelization Issues

Since our target parallel architecture is the GPU, we briefly introduce its architecture and its consequences on DE simulation programming. The Graphics Processing Units (GPUs) are high-performance many-core coprocessors initially designed to accelerate graphics rendering. They are increasingly used to accelerate general purpose scientific and engineering computing (GPGPU). The GPU, or *device*, is controlled by a CPU, also called *host*. The GPU architecture being significantly different form the CPU, we first introduce its most important characteristics necessary to understand the design and implementation of the GPU specific DEM algorithm we propose. We then present CUDA, the *de facto* standard GPGPU programming environment. This allows us to finally discuss the parallelization of the DE simulation.

3.1. *GPU Architecture Overview*

The GPU is built from multiple computing cores called multiprocessors. Each multiprocessor is an array of synchronous scalar processor units enabling concurrent SIMD computations. SIMD stands for Single Instruction Multiple Data streams, meaning that all scalar processors in a multiprocessor can only execute the same instruction, on different data, at a given time. It is well adapted for data-parallel applications, where a large amount of data undergoes similar computations. Its limitation is that when executing a conditional block, all scalar processors evaluating the condition to true proceed synchronously with the block execution while the other scalar processors are kept idle. Obviously, full efficiency is achieved when all scalar processors follow the same execution path.

Multiprocessors have less double precision floating point units than single precision ones. Thus, performance can significantly drop when moving floating operations from single to double precision. For example, recent GPUs from the NVIDIA Tesla family feature up to 665 GigaFlops of double precision perfomance and 1 TeraFlops of single precision performance (the ratio single/double was about 8 in previous generations).

The GPU manages its own memory, separate from the CPU. Data transfers between the CPU and GPU are explicitly controlled by the application. The GPU memory is divided between a global memory all multiprocessors can access, and a shared memory local to each multiprocessor. Data transfers between global and shared memory are also explicit. GPUs also have a read-only global memory (constant memory), one part having special access functions (texture memory), but we will not detail these memories which we do not use in this work.

When the data accessed by different threads of a multiprocessor is contiguous, these accesses are coalesced, i.e. the data is gathered to be transfered in a single pass. Otherwise they are performed sequentially, incurring a strong performance penalty.

Concurrent writings to the same address can lead to inconsistencies. It is up to the programmer to avoid this, or to use *atomic* operations. An operation is said *atomic* when it can not be interrupted by any concurrent processes before it ends up. Atomic operations are transparently sequentialized by the GPU, at the price of a loss of efficiency.

Each multiprocessor has a limited number of registers shared by scalar processors. Only scalar processors that can have their register needs filled will be able to perform computations, the other will be idle. Register use must therefore be carefully managed to optimize the performance.

3.2. *GPU Programming Overview*

NVIDIA is currently the leader in GPGPU, and we therefore focus on their architecture and associated programming environnement, CUDA (NVIDIA Corporation, 2011). The implementation presented in this paper could be ported to the OpenCL emerging standard for GPGPU programming without major difficulty (OpenCL encompasses the CUDA programming model), with probably lower performance, as OpenCL compilers still need to gain in maturity.

The CUDA programming model closely matchs the GPU architecture. The base parallelization unit is a *thread*, which is a sequence of instructions executed on a scalar processor. The CUDA programmer writes programs to be executed on a scalar processor, also called *kernels*. The kernels are transfered to the GPU and their parallel execution is parameterized by *block* sizes. A block is a group of threads executed by the same multiprocessor. Inside a block, each thread is identified by a unique index. The address of the thread data in the memory is easily computed using the block and thread indices. When the number of threads per block is greater than the actual number of scalar processors, the execution is composed of a sequence of *warps*. Each warp contains one thread per scalar processor. The number of threads per block should thus be a multiple of the warp size. If the active threads become idle waiting for data from the memory, they will be transparently suspended to allow an other warp to become active, enabling to overlap computations with memory accesses. All threads within the same warp are executed synchronously as imposed by the SIMD architecture of the multiprocessor. Communications between the threads belonging to the same block are enabled through memory sharing and synchronization instructions. Threads in different blocks can not directly interact.

3.3. *Parallelization of the DE Simulation*

In this section, we review the main phases of the DE simulation loop and we discuss the parallelization issues due to the GPU architecture, especially the concurrent writing. The simulation relies on an explicit integration scheme. The integration loop (Figure 1) can be divided in three main parts. In the first part, the positions and ve-

locities of the DEs are updated based on known forces using Newton's laws of motion within a given timestep. The parallelization is straightforward, each DE being processed independently.

The second part consists in detecting new contacts between the DEs, which interact if the distance between their centroids is less than the sum of their interaction radii. This can be compute intensive, as the number of potential contact pairs grows quadratically along with the number of DEs. A common approach to accelerate the collision detection process is to define a 3D grid that divides the space filled by DEs into cubic cells, with size equal to the maximum interaction diameter of all DEs. The first collision detection step consists in identifying the cell each DE belongs to. Then for each DE we search for new contacts browsing other DEs from the cell it belongs to and from the 26 neighbor cells. This is again data-parallel, but appending a new contact to the list requires a concurrent access. To avoid duplicating contacts, we need for each element an up-to-date list of every other DE it interacts with. Managing this dynamic set of links significantly complexifies the implementation. Without altering the simulation correctness, the collision detection can be executed periodically and not at every iteration. This period depends on the simulation parameters. In our simulations, collision detection is typically scheduled every 10 time step.

The last part is force computation and link update. Each force computation is costly, as the high number of parameters required to represent the concrete structure behavior at a macroscopic scale leads to numerous operations. The interaction force can be computed in parallel on each link. However, the accumulation of forces on each DE requires a concurrent access to the force vector of each DE. We then detect the links and contacts that break due to excessive distance or interaction force. This is straightforwardly parallel, but the list of contacts needs to be updated. The initial links can be marked as broken, to avoid inefficient list compactions.

Finally, since stability is a major issue with explicit integration schemes, the time step is dynamically adjusted at the end of every step (Subsection 4.4). Using DEs' stiffnesses for translation and rotation, we compute interactions stiffnesses that are then accumulated for each DE, which requires another concurrent write operation. The time step is then set to the minimum over every coordinates of the square root of the ratio of inertia over stiffnesses.

3.4. *Comparison with Molecular Dynamics*

DEM simulations are close to Molecular Dynamics (MD) simulations, notably because both are dealing with moving particles, with interactions between particles leading to forces, as well as periodic collision detections (neighbor search in MD). Anderson *et al.* (2008) proposed a GPU implementation for MD, later improved by Rapaport (2011), for example using an interaction cutoff range for the dimension of the detection grid during the neighbor search. Though their implementation shows several similarities with the one we propose in this paper, there remain important

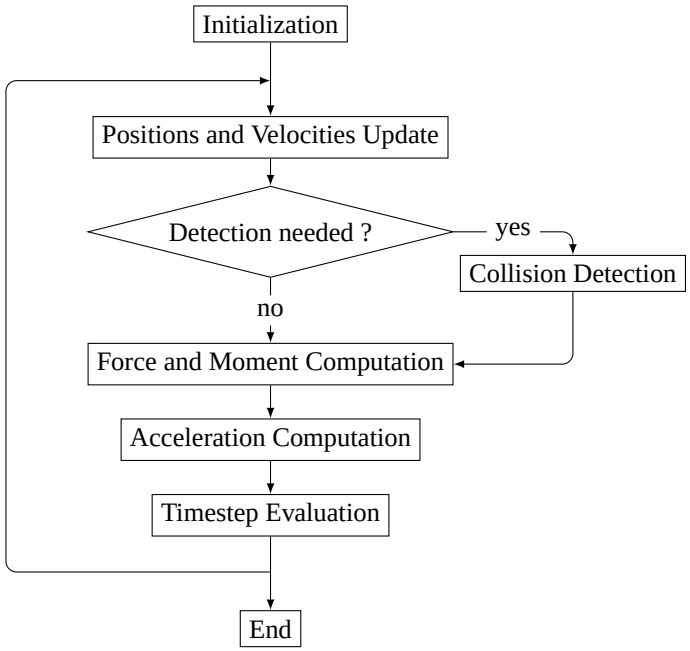


Figure 1: Global simulation integration scheme.

differences. First, interactions between DEs have a stateful history while forces corresponding to interacting MD particles observe some well defined laws. Thereby, in MD simulations when the neighbor list is updated there is no need to unselect already existing interactions, which requires storing the list of existing interactions. Moreover DEM computations are more complex, since DEs have rotational degrees of freedom, and are of different types (steel, concrete), requiring different interaction types. Consequently, DEM computations are so greedy in term of operations and registers need that it is efficient to compute one force per interaction, and then to accumulate these forces at each DE. Conversely, because MD is significantly less computationally intensive, it is more efficient on GPU to compute all interaction forces for each atom (each force is thus computed twice).

4. Implementation on the GPU

We now detail our GPU implementation composed of a sequence of 19 CUDA kernels (Figure 2). Many of these kernels perform data parallel operations running one thread per link, contact or DE. The implementation challenge is to manage memory accesses as well as double precision computations. For that purpose data is maintained in special structures in the global memory to ensure correct alignment in memory to favor coalesced parallel accesses.

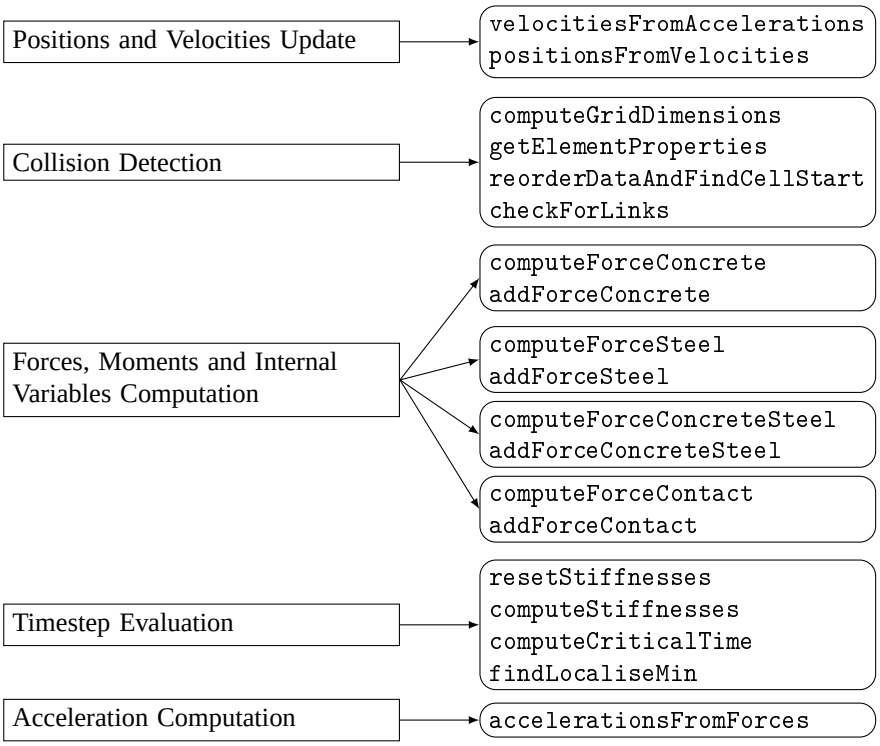


Figure 2: CUDA kernels related to integration steps.

4.1. *Memory Layout*

Let B be the number of threads per block and N the number of DEs. All state vectors (position, velocity, forces and accelerations) are stored per block of B elements. All the vectors have 3 or 6 components. For instance a force has six coordinates, three for centroids positions and three for the element rotations. The $6 \times N$ force components are stored continuously in memory, with padding data at the end to obtain a $\lceil \frac{6 \times N}{B} \rceil \times B$ size array (Figure 3). Within a block, all threads fetch the force components from the GPU global memory into the shared memory by series of B contiguous, aligned, concurrent and thus coalesced memory accesses. Next, after a synchronization to ensure all memory transfers are effective, each thread loads from the shared memory the data it needs for computations. At the end of the kernel, the updated forces stored in shared memory are written back to the global memory following the same pattern. This data organization ensures efficient memory transfers while keeping data conveniently organized in the GPU global memory if the CPU needs to access them.

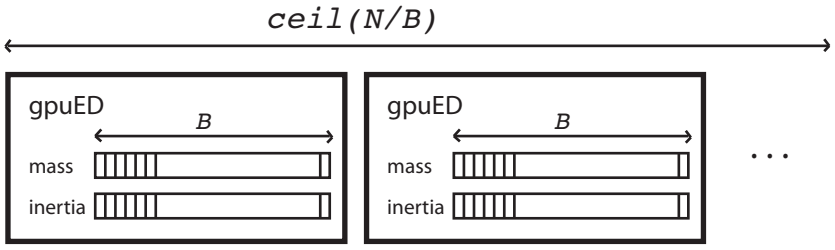


Figure 3: GPU structure layout for DE masses and inertia. An array of B elements masses is associated with an array of B elements inertia in a structure called $gpuED$. The whole set is constructed by an $\lceil \frac{N}{B} \rceil$ length array of these structures.

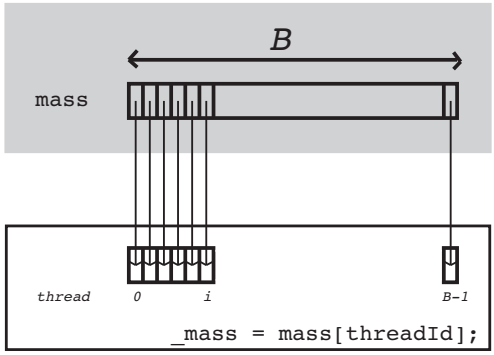


Figure 4: Illustration of a data load from global memory (grey zone) to thread registers. Every thread within a warp loads its value and this is done in a single coalesced memory transfer.

The rest of the internal data is more directly stored in global GPU memory in arrays of B size blocks. For instance the `accelerationsFromForces` kernel computes accelerations using masses and inertia. For sake of commodity and to favor data locality, masses and inertia are stored in a single array interleaving B masses and B inertia as illustrated in figure 3. Thus the threads of a block access masses or inertia data taking advantage of the GPU coalescing capabilities (Figure 4). In this case, the data is directly loaded to registers without using the shared memory. This layout is applied to every set of internal variables stored in the GPU global memory and used only by the device.

4.2. *Force Computation*

A series of kernels are in charge of force computations (`computeForce*`). Each kernel takes care of one interaction type, operating on links or contacts only. We adopted this multi-kernel organization to avoid the divergent conditional branches required to support different interaction types. The amount of data parallelism available would have been reduced. Next, forces are accumulated on each DE, a very data parallel operation the `addForce` takes care of. In a recent paper, Shigeto *et al.* (2011) directly accumulate computed force contributions using an atomic add operation. However this operation is costly. Though our approach requires to maintain the list of interactions each DE is involved in, it does not require any atomic operation.

4.3. *Collision Detection*

Collision detection is performed using four CUDA kernels. `computeGridDimensions` computes the bounding box of the current simulation state. `getElementPositionInGrid` identifies the cell each DE belongs to. The third one, `reorderDataAndFindCellStart`, sorts DEs according to cell indices and computes the memory offset to identify for each cell the starting address of its list of DEs. Then the kernel `checkForContacts` detects the new contacts. More precisely, the first step computes the dimension of the bounding box that includes all DEs in the three-dimensional space. This is done by an optimized parallel search of both maximum and minimum among DE positions . The cell size is set to a multiple of the size of the maximum interaction radius in every of the three dimensions. Next, a kernel computes the cell index of each DE. We store a cell index as a key and the DE index as the value in two corresponding arrays. Next, we sort the DE indices according to their cell index using the parallel radix sort operation from the CUDPP library (Satish *et al.*, 2009). The size of these structures is proportional to the number of DEs and not to the grid resolution. To avoid processing the whole cell array searching for the neighbors of a given DE, an other kernel builds an indirection table giving for each cell the beginning and end index of its DE list. To improve the local coherence when searching for neighbors, DE positions and radii are also sorted into new structures according to their particle order. This method has been implemented by NVIDIA in a simple demo called Particles in the CUDA SDK (NVIDIA, 2010), although they do not need to compute the bounding box at each step as the simulation space does not change during the simulation.

Everything being set up, the interaction search can begin ! Search is parallelized over DEs. Every thread is in charge of testing new interactions with the particles present in its cell and neighbor cells. To avoid redundant computations, only DEs with higher indices than the current DE are tested in the DE's cell, and only half of the neighbor cells are tested. We also maintain the neighbor list of each DE, used to check if DEs are already connected together (either with an initial link or a previously detected contact). When a new interaction is detected, it is pushed into a list.

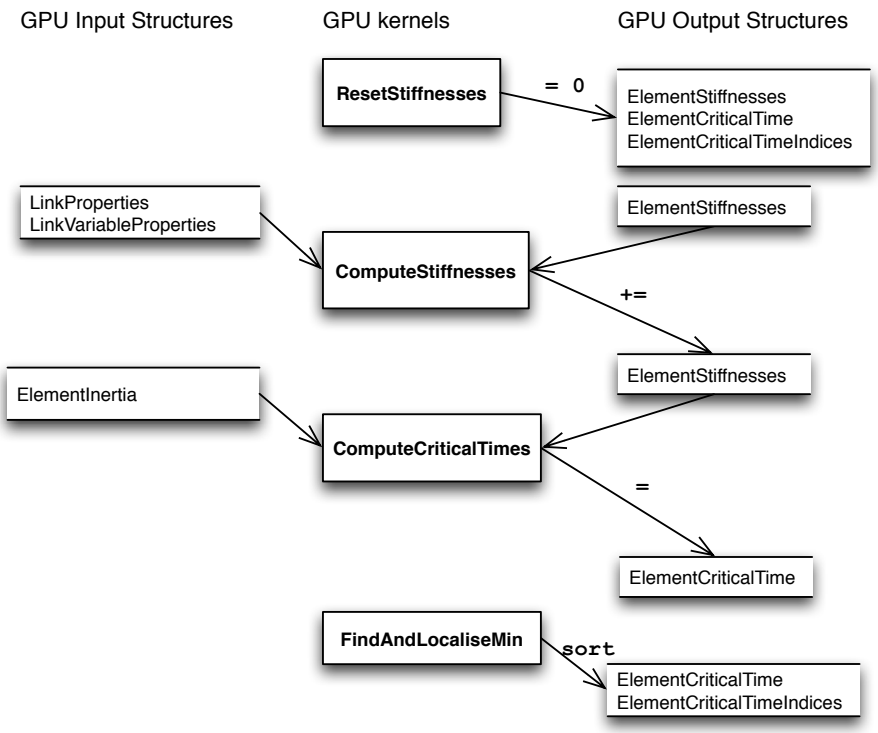


Figure 5: Detailed process for checkTimestep with CUDA kernels and Cuda structures names.

4.4. *Time step*

To compute the new time step (Figure 5), we first reset the values computed during the previous step with a call to `resetStiffnesses` relying on the GPU `memset` function. Stiffnesses are computed per interaction type by `computeStiffnesses` and cumulated in the corresponding elements stiffnesses. Then the `computeCriticalTimes` kernel computes a local critical time T_c per DE from the 6 stiffnesses (three position and three rotation components) and the inertia. This kernel ends by executing a parallel minimum search to obtain the smallest critical time, i.e. the time step for the next step.

Finally, accelerations are computed based on forces by the `accelerationsFromForces` kernel, followed by an update of velocities and positions (`velocitiesFromAccelerations` and `positionsFromVelocities`).

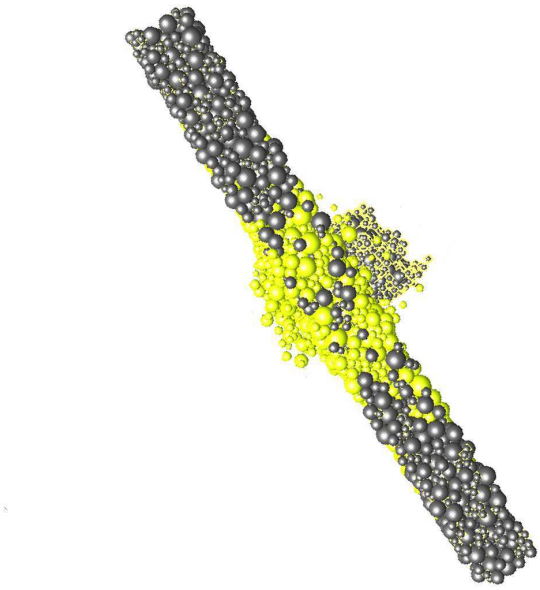


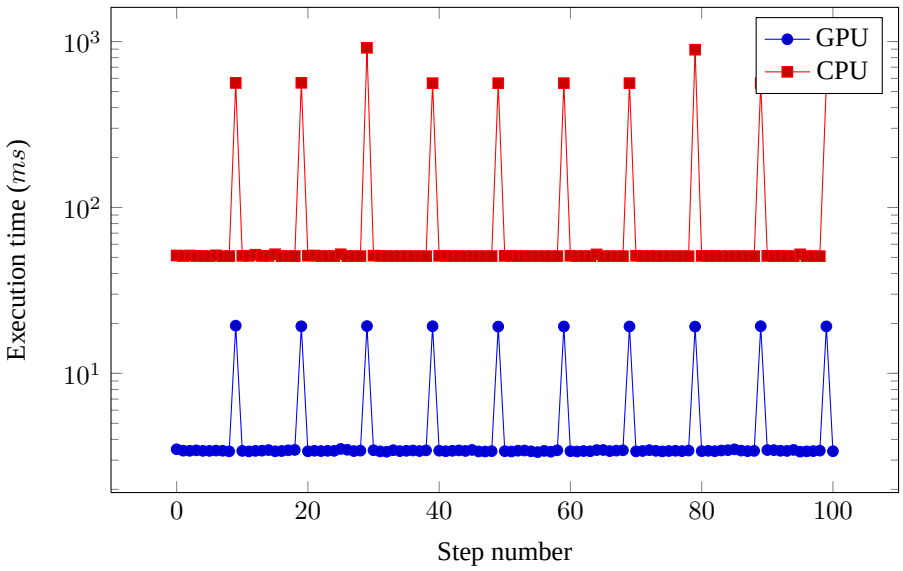
Figure 6: Simulation of an impactor projected against a reinforced concrete slab. The picture has been taken at step 2340, during the impact itself.

Attention must be paid to floating-point operations. Depending on the GPU and the CUDA version used, default operators are not IEEE-754 compliant and must be changed for IEEE-754 compliant ones. In our case kernels are implemented using IEEE-754 with round to nearest compliant operations and can be compiled to support either single or double precision floating point data.

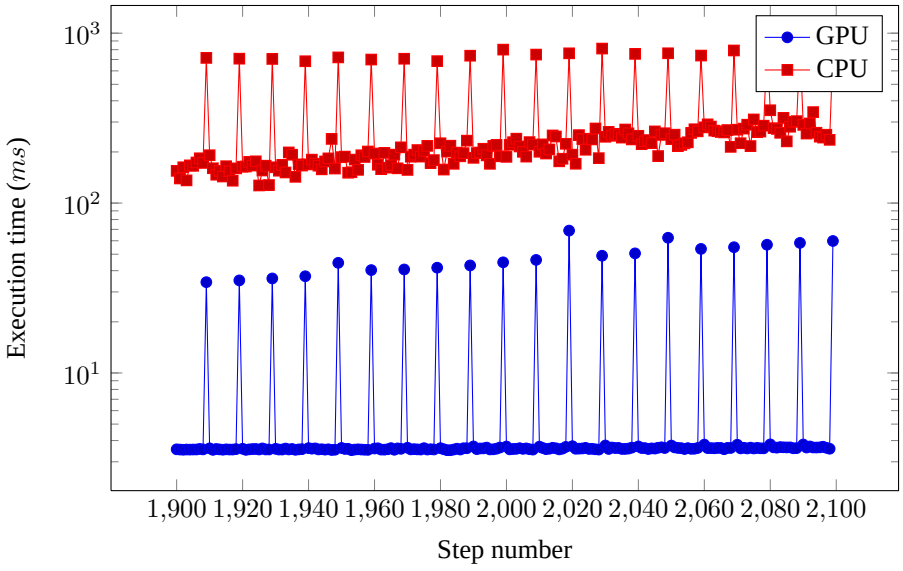
5. Results

We measured the performance of our implementation for the simulation of an impact on a reinforced concrete slab (Figure 6). The scene involves $14,274$ DEs with 784 being reinforcing steel rods, $12,718$ the slab concrete and 772 the concrete representing the impactor. The slab dimensions are 1m x 1m x 0.1m. There are 79211 cohesive links at start. The GPU used is a NVIDIA Fermi C2050 with 448 cores and 3Go of memory while the CPU is a 6-core Intel Xeon X5650 @2.67GHz (only one core is used for running the simulation).

Figure 7a presents an example of each step execution time at different stages of the simulation. This plot figures the starting of the simulation. The peaks correspond to the steps performing the collision detection. At the beginning, execution times



(a) One hundred steps, before the impactor touches the slab.



(b) Two hundred steps, the impactor has reached the slab.

Figure 7: Execution time (ms - logarithmic scale) per step on the GPU or the CPU. Time peaks occur when the collision detection is performed.

are constant because the impactor is not yet colliding with the slab. The collision detection execution is 10 times slower than the other calculation stages on the CPU (6 times on GPU). Thus, when performed every 10 steps, it takes about 50% of the average execution time on the CPU (70% on the GPU). The speed-up reached by our GPU implementation is about 15 when the collision detection is not performed and about 30 otherwise compared to the execution time on the GPU.

Results during the impact are presented to show the evolution of the simulation behavior (Figure 7b). Considering steps without collision detection, on the CPU, the execution time increases during the simulation because of the creation and removal of interactions. On the GPU, the execution time appears constant. The data structures for initial interactions are not altered during the simulation, broken links being simply invalidated, which does not significantly change the execution time needed to launch and perform the kernels for these interactions. The execution time can increase when contacts are created. However, as these extra contacts are processed in parallel, their number needs to be important to have a significant impact on the overhall execution time.

During the collision detection on the CPU, the particles are sorted in cells using linked lists. Dynamic memory allocation can lead to an execution time overhead, but this is efficient regarding memory space, and execution time is not affected by the spreading of particles in space. Conversely, on the GPU, the parallelism is reduced when particles start being fired away so the collision detection becomes less efficient. As a consequence, the execution time on the GPU tends to increase for collision detections.

6. Conclusion

We presented how the DEM can be used to simulate concrete structures on GPU. After a reminder of the method and its parametrization, we detailed our GPU implementation and simulation results. One challenge of the DEM is its high computational cost. However, its inherent data parallel structure and limited memory footprint make it a good candidate to take advantage of the high performance of GPUs. Our experiments show a speed-up of an order of magnitude for 14274 particles compared to an execution on a single core CPU.

This implementation work revealed several difficulties. A careful management of memory access patterns is essential to ensure that the GPU processing units do not stall waiting for data. The necessity of managing 2 different types of particles, for concrete and for steel, reduces the data parallelism of the application and almost duplicates the number of GPU kernels to execute. Only simulations with a high particle count amortize the associated overheads. The neighbor computation step that is required to identify the interacting particles exhibits a limited level of data parallelism that, creates a performance bottleneck, even using optimized sorting GPU implementations.

The high implementation cost may counterbalance the performance gain, especially given today's availability of CPUs with up to 12 cores. A careful parallel implementation on such processors could probably lead to a significant speed-up. Moreover, good tools for debugging and monitoring GPU implementations are still missing, making the development painful. However GPGPU is still in its early stage and is quickly evolving. The next GPU generation will provide features that may give a significant advantage to GPGPU. The Intel MIC coprocessor (Seiler *et al.*, 2008) has a large number of x86 cores (32 for the current development prototype), each core having a large vector unit for data parallel operations. This architecture will offer a higher level of flexibility, enabling cores to work asynchronously, locally having to extract only a limited amount of data parallelism to fill the vector unit, while globally showing a similar amount of parallel processing power. Because the MIC relies on a classical x86 architecture, we can expect to benefit from the development tools already available for x86 processors. The AMD APU (Advanced Processing Unit) integrates on a single chip a CPU and a GPU. It will enable the CPU and the GPU to share the same memory, significantly decreasing the cost of data exchange between the CPU and the GPU.

7. References

Anderson J. A., Lorenz C. D., Travesset A., " General purpose molecular dynamics simulations fully implemented on graphics processing units", *J. Comput. Phys.*, vol. 227, p. 5342-5359, May, 2008.

Camborde F., Mariotti C., Donzé F. V., " Numerical study of rock and concrete behaviour by discrete element modelling", *Computers and Geotechnics*, vol. 27, n 4, p. 225 - 247, 2000.

Cundall P. A., Strack O. D. L., " A discrete numerical model for granular assemblies", *Geotechnique*, vol. 29, n 1, p. 47-65, 1979.

D'addeta G. A., Kun F., Ramm E., " On the application of the discrete model to fracture propagation in concrete.", *Granulat Matter*, vol. 4, p. 77-90, 2002.

Dhia H. B., Rateau G., " The Arlequin method as a flexible engineering design tool", *International Journal for Numerical Methods in Engineering*, vol. 62, n 11, p. 1442-1462, 2005.

Frangin E., Marin P., Daudeville L., " On the use of combined finite/discrete element method for impacted concrete structures", *J. Phys. IV France*, vol. 134, p. 461-466, 2006.

Hentz S., Daudeville L., Donzé F. V., " Identification and Validation of a Discrete Element Model for Concrete", *Journal of Engineering Mechanics*, vol. 130, n 6, p. 709-719, 2004.

Lee V. W., Kim C., Chhugani J., Deisher M., Kim D., Nguyen A. D., Satish N., Smelyanskiy M., Chennupaty S., Hammarlund P., Singhal R., Dubey P., " Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU", *SIGARCH Comput. Archit. News*, vol. 38, p. 451-460, June, 2010.

NVIDIA Corporation, " NVIDIA CUDA Compute Unified Device Architecture Programming Guide", , CUDA Documentation, June, 2011.

NVIDIA S. G., " Particle Simulation Using CUDA", , CUDA SDK, May, 2010.

Potapov S., Faucher V., Daudeville L., “ Advanced simulation of damage of reinforced concrete structures under impact”, *European Journal of Environmental and Civil Engineering*, 2012.

Rapaport D., “ Enhanced molecular dynamics performance with a programmable graphics processor”, *Computer Physics Communications*, vol. 182, n 4, p. 926 - 934, 2011.

Rousseau J., Frangin E., Marin P., Daudeville L., “ Damage prediction in the vicinity of an impact on a concrete structure: a combined FEM/DEM approach.”, *Computers and Concrete*, vol. 5, n 4, p. 343-358, 2008.

Rousseau J., Frangin E., Marin P., Daudeville L., “ Multidomain finite and discrete elements method for impact analysis of a concrete structure”, *Engineering Structures*, vol. 31, n 11, p. 2735 - 2743, 2009.

Rousseau J., Marin P., Daudeville L., Potapov S., “ A discrete element/shell finite element coupling for simulating impacts on reinforced concrete structures”, *European Journal of Computational Mechanics*, 2010.

Satish N., Harris M., Garland M., “ Designing efficient sorting algorithms for manycore GPUs”, *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, p. 1 -10, may, 2009.

Seiler L., Carmean D., Sprangle E., Forsyth T., Abrash M., Dubey P., Junkins S., Lake A., Sugerman J., Cavin R., Espasa R., Grochowski E., Juan T., Hanrahan P., “ Larrabee: a many-core x86 architecture for visual computing”, *ACM Trans. Graph.*, vol. 27, p. 18:1-18:15, August, 2008.

Shigeto Y., Sakai M., “ Parallel computing of discrete element method on multi-core processors”, *Particuology*, vol. In Press, Corrected Proof, p. -, 2011.

Tran V., Donzé F.-V., Marin P., “ A discrete element model of concrete under high triaxial loading”, *Cement and Concrete Composites*, vol. In Press, Corrected Proof, p. -, 2011.

Xiao S. P., Belytschko T., “ A bridging domain method for coupling continua with molecular dynamics”, *Computer Methods in Applied Mechanics and Engineering*, vol. 193, n 17-20, p. 1645 - 1669, 2004. Multiple Scale Methods for Nanoscale Mechanics and Materials.