J. Parallel Distrib. Comput. 69 (2009) 451–460

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

# Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA

Dimitri Komatitsch [a,b,*], David Michéa [a], Gordon Erlebacher [c]

[a] Université de Pau et des Pays de l'Adour, CNRS & INRIA Magique-3D, Laboratoire de Modélisation et d'Imagerie en Géosciences UMR 5212, Avenue de l'Université, 64013 Pau Cedex, France

[b] Institut universitaire de France, 103 boulevard Saint-Michel, 75005 Paris, France

[c] Department of Scientific Computing, Florida State University, Tallahassee 32306, USA

ARTICLE INFO

*Article history:*
Received 30 June 2008
Received in revised form
25 October 2008
Accepted 16 January 2009
Available online 3 February 2009

*Keywords:*
GPGPU
CUDA
Speedup
Finite elements
Spectral methods

ABSTRACT

We port a high-order finite-element application that performs the numerical simulation of seismic wave propagation resulting from earthquakes in the Earth on NVIDIA GeForce 8800 GTX and GTX 280 graphics cards using CUDA. This application runs in single precision and is therefore a good candidate for implementation on current GPU hardware, which either does not support double precision or supports it but at the cost of reduced performance. We discuss and compare two implementations of the code: one that has maximum efficiency but is limited to the memory size of the card, and one that can handle larger problems but that is less efficient. We use a coloring scheme to handle efficiently summation operations over nodes on a topology with variable valence. We perform several numerical tests and performance measurements and show that in the best case we obtain a speedup of 25.

## 1. Introduction

Over the past several years, non-graphical applications ported to the GPUs have steadily grown more numerous [25]. Many applications have been adapted to the GPU using one of several specialty languages. These range from graphical languages such as Cg, HLSL, GLSL, to languages that abstract away the graphics component to ease the implementation of non-graphical code, such as Brook, Scout [20], Sh, Rapidmind, and CUDA. CUDA offers significant innovation and gives users a significant amount of control over a powerful streaming/SIMD computer. Since CUDA became available in 2007, the number of General-purpose Processing on Graphical Processing Units (GPGPU) applications has drastically increased. CUDA is being used to study physical problems as diverse as molecular dynamics [2], fluid dynamics [4], or astrophysical simulations [24].

Our research group has developed a software package, called SPECFEM3D_GLOBE, for the three-dimensional numerical simulation of seismic wave propagation resulting from earthquakes in the Earth based on the so-called spectral-element method (SEM) [16,19,17]. The SEM is similar to a finite-element method with high-degree polynomial basis functions. The mesh of elements required to cover the Earth is usually so large that calculations are very expensive in terms of CPU time and require a large amount of memory, at least a few terabytes. We therefore usually turn to parallel programming with MPI [17,15], OpenMP, or both. Typical runs require a few hundred processors and a few hours of elapsed wall-clock time. Current large runs require a few thousand processors, typically 2000–4000, and two to five days of elapsed wall-clock time [17,15]. It is therefore of interest to try to speed up the calculations on each node of such a parallel machine by turning to GPGPU. We will see in the next section that a significant amount of work has been devoted in the community to porting low-order finite-element codes to GPU, but to our knowledge we are the first to address the issue of porting a more complex higher-order spectral-like finite-element technique.

Rather than attempt to solve the fully resolved problem using both MPI and the GPU simultaneously, which we will consider in a future study, we choose to investigate the potential for speedup on a single CPU, with acceleration provided solely by the GPU. Such an implementation can be very useful in the absence of a GPU cluster for quick parameter studies, parametric explorations, and code development. In the future the experience gained by this exercise will provide the proper intuition to accelerate an MPI version of the

[*] Corresponding author at: Université de Pau et des Pays de l'Adour, CNRS & INRIA Magique-3D, Laboratoire de Modélisation et d'Imagerie en Géosciences UMR 5212, Avenue de l'Université, 64013 Pau Cedex, France.

*E-mail addresses:* dimitri.komatitsch@univ-pau.fr (D. Komatitsch), davidmichea@gmail.com (D. Michéa), gerlebacher@fsu.edu (G. Erlebacher).

*URLs:* http://www.univ-pau.fr/~dkomati1 (D. Komatitsch), http://www.sc.fsu.edu/~erlebach (G. Erlebacher).

0743-7315/$ – see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2009.01.006

code using the graphics processors. Our seismic wave propagation application is written in Fortran95 + MPI, but we wrote a serial version in C for the tests to facilitate interfacing with CUDA, which is currently easier from C.

A major issue in all finite-element codes, either low or high order, is that dependencies arise from the summation of elemental contributions at those global points shared among multiple mesh elements, requiring atomic operations. We will show that use of a mesh coloring technique allows us to overcome this difficulty and obtain a speedup of 25x, consistent with speedups obtained by many other researchers for other realistic applications in different fields.

The remainder of the paper is organized as follows: We begin with a description of similar problems ported to the GPU in Section 2. The serial algorithm is discussed in Section 3, followed by the CUDA implementation of the spectral-element algorithm in Section 4. Within this section, we discuss an implementation that runs fully on the GPU, including the coloring scheme that is needed to avoid atomic operations. We also describe a version of our algorithm capable of solving larger problems that cannot fit fully on the GPU. Optimizations we considered to improve efficiency are treated in Section 5. We provide some numerical code validation in Section 6, and analyze some performance results in Section 7. We conclude in Section 8.

## 2. Related work

For a wide range of applications, depending on the algorithms used and the ease of parallelization, applications ported to the GPU have acquired speedups that range from 3x to 50x. Of course, whether a specific speedup is considered impressive or not depends strongly on how well the original code was optimized. Algorithms such as finite-difference and finite-volume methods are amenable to large accelerations due to their sparse structure. On the other hand, dense methods such as spectral methods or spectral finite-element methods are more difficult to accelerate substantially, partly due to their dense matrices combined with the small amount of fast memory on the graphics card. Thus, many codes become bandwidth-limited. On the other hand, the local nature of a finite-difference stencil allows smaller sections of the domain to be treated at a given time, and there is therefore more room to balance issues of bandwidth, computations, kernel size, etc. [14,11].

Göddeke and his co-workers have successfully implemented their finite-element algorithms on a CPU cluster enhanced by now out-of-date GPUs. They program the GPUs at the shader level to solve an implicit system using a multigrid algorithm [9]. In the area of numerical modeling of seismic wave propagation, [1] has recently successfully harnessed GPGPU to calculate seismic reverse-time migration for the oil and gas industry using a finite-difference method.

To date, there have been a few finite-element implementations in CUDA, such as the volumetric finite-element method to support the interactive rates demanded by tissue cutting and suturing simulations during medical operations [29,30,25]. The acceleration derives from a GPU implementation of a conjugate gradient solver. The time-domain finite-element has been accelerated using OpenGL on the graphics card [18]. The authors have achieved a speedup of only two since the code is dominated by dense vector/matrix operations.

To our knowledge, the first nonlinear finite-element code ported to the GPU is in the area of surgical simulation [27]. The finite-element cells are tetrahedra with first-order interpolants and thus the overall solver is second-order accurate. The authors achieve a speedup of 16x. The algorithm is implemented fully on the GPU using both graphic and non-graphic APIs. The authors decided not to use one of the existing higher-level languages such as CUDA for NVIDIA graphics cards or CTM for ATI graphics cards. The unique structure of the tetrahedra allows the storage of the force at the nodes of a tetrahedra in four textures of a size equal to the number of global nodes. Once these forces are calculated, a pass through the global nodes combined with indirect addressing allows the global forces to be calculated. In our code, a novel coloring scheme decomposes the mesh into distinct subsets of elements with no common vertices within a subset. The global forces can thus be calculated in multiple blocks without any risk of interaction. This simplifies the data structures tremendously, and speeds up the code.

In [28], the authors invert a mass matrix with Jacobi iteration using C# and a Microsoft interface to the GPU. The remainder of the code is solved on the GPU. Although the authors achieve a speedup of close to 20x, the Jacobi method is intrinsically parallel, and has a very slow convergence rate.

For real and relatively complex applications such as ours, it is typically expected to obtain a speedup between 10x and 20x when turning to GPGPU. For instance, for a wave-equation application, i.e., solving the same physical equation as in this article, as well as for other problems such as a rigid-body physics solver or matrix numerics, NVIDIA itself reports speedups between 10x and 15x ([5], page 26 of the PDF presentation). Table 2 of [6] also measures speedups around 10x to 15x for six real applications, and 44x for a seventh. For other problems such as sorting, fast Fourier transform and dense matrix multiplication algorithms, [11] have developed cache-efficient algorithms on GPU and also obtained speedups ranging between 2x and 10x.

Current GPGPU hardware supports quasi-IEEE-754 s23e8 single precision arithmetic, except for instance the fact that denormalization is lacking and always rounds to zero [10]. A weak point of some current graphics cards is their lack of hardware support for 64 bit double precision arithmetic. Recent NVIDIA GPUs such as the G200 series support double precision, but, according to NVIDIA, with a peak speed of 90 Gflops [23], a full order of magnitude lower than its peak speed of 988 Gflops in single precision.

For some other finite-element applications, which involve the resolution of large linear systems, people have been forced to turn to mixed 'single-double' precision solvers or to emulate double precision in software [10]. Our SEM code is sufficiently accurate in single precision, see for instance [16] and benchmarks therein, as well as the example in the section about numerical results below, and therefore current hardware is sufficient.

It is worth mentioning that another approach, called 'automatic speculative thread extraction (ASTEX) technique for hybrid multicore architectures', has been developed by INRIA CAPS [26,7] to port existing applications to GPGPU without the need to modify source code. Runtime data is used to automatically extract parts of an existing C code and provide speculative information on data structure accesses to partition the code between general-purpose cores and hardware accelerators such as GPUs, FPGAs or coprocessors. The partitioning of the application source code is then expressed through directives called HMPP. Another library, called GPULib [21], is available for IDL, Matlab and Python based on a similar idea: hiding the difficulty of programming GPUs directly by providing a high-level library.

## 3. Serial algorithm

The SEM is a high-order finite-element algorithm in which a domain is subdivided into cells within which variables are approximated with high-order interpolation. We have adopted the SEM to simulate numerically the propagation of seismic waves resulting from earthquakes in the Earth [16]. It solves the variational form of the elastic wave equation in the time domain on a non-structured mesh of hexahedral elements called spectral elements to compute the displacement vector of any point of the medium under study, knowing the density and the speed of pressure and shear waves (or equivalently, Lamé parameters) in the rocks that compose it.

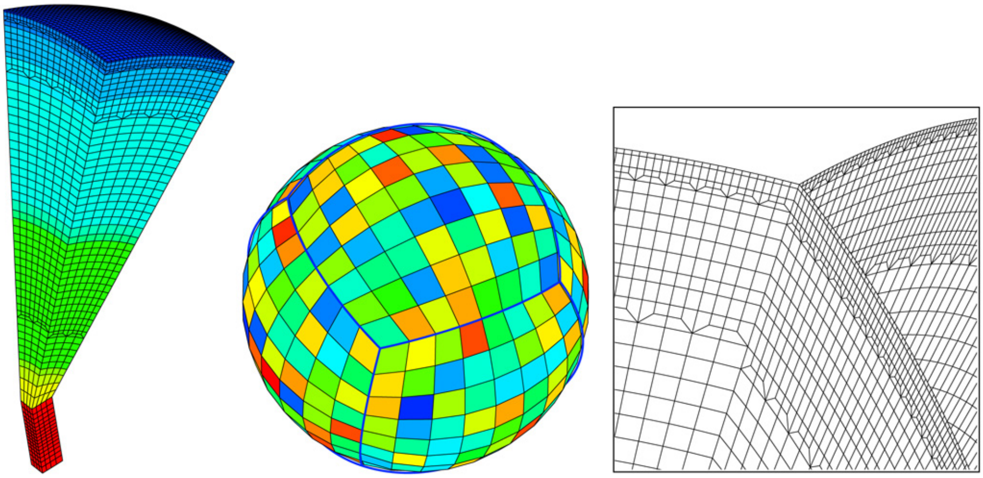

**Fig. 1.** (Left) A typical mesh slice extracted from our MPI spectral-element seismic wave simulation software package for the full Earth. The full mesh for the MPI version contains hundreds or thousands of such mesh slices in order to mesh the full sphere (middle, in which each slice is shown with a different color and the top of each mesh slice can be seen at the surface of the sphere). In this article we use only a single slice to make the code serial and to enable testing of speedup obtained with CUDA on the GPU. Each so-called spectral element of the non-structured mesh contains $5^3 = 125$ grid points not represented here, but represented in Fig. 2. (Right) Close-up on 2D cut planes, showing that the mesh does not have a regular topology: it is non-structured.

We consider a linear anisotropic elastic rheology for the heterogeneous solid Earth, and therefore the seismic wave equation can be written in the strong, i.e., differential, form

$$\begin{aligned} \rho\ddot{\mathbf{u}} &= \nabla\cdot\boldsymbol{\sigma} + \mathbf{f}, \\ \boldsymbol{\sigma} &= \mathbf{C} : \boldsymbol{\varepsilon}, \\ \boldsymbol{\varepsilon} &= \frac{1}{2}[\nabla\mathbf{u} + (\nabla\mathbf{u})^{\mathrm{T}}], \end{aligned} \tag{1}$$

where $\mathbf{u}$ denotes the displacement vector, $\boldsymbol{\sigma}$ the symmetric, second-order stress tensor, $\boldsymbol{\varepsilon}$ the symmetric, second-order strain tensor, $\mathbf{C}$ the fourth-order stiffness tensor, $\rho$ the density, and $\mathbf{f}$ an external force. The double tensor contraction operation is denoted by a colon, a superscript T denotes the transpose, and a dot over a symbol indicates time differentiation. The physical domain of the model is denoted by $\Omega$ and its outer boundary by $\Gamma$. The material parameters of the solid, $\mathbf{C}$ and $\rho$, can be spatially heterogeneous. We can then rewrite the system (1) in a weak, i.e., variational, form by dotting it with an arbitrary test function $\mathbf{w}$ and integrating by parts over the whole domain:

$$\begin{aligned} &\int_{\Omega} \rho\,\mathbf{w}\cdot\ddot{\mathbf{u}}\,\mathrm{d}\Omega + \int_{\Omega} \nabla\mathbf{w} : \mathbf{C} : \nabla\mathbf{u}\,\mathrm{d}\Omega \\ &\quad = \int_{\Omega} \mathbf{w}\cdot f\,\mathrm{d}\Omega + \int_{\Gamma} (\boldsymbol{\sigma}\cdot\hat{\mathbf{n}})\cdot\mathbf{w}\,\mathrm{d}\Gamma. \end{aligned} \tag{2}$$

The free surface boundary condition, i.e., the fact that the traction vector must be zero at the surface, is easily implemented in the weak formulation since the integral of traction $\boldsymbol{\tau} = \boldsymbol{\sigma}\cdot\hat{\mathbf{n}}$ along the boundary simply vanishes when we set $\boldsymbol{\tau} = \mathbf{0}$ at the free surface of the Earth.

This formulation is solved on a mesh of hexahedral elements in 3D, which honors both the free surface of the model and its main internal discontinuities, i.e., its geological layers. The unknown wave field is expressed in terms of Lagrange polynomials of degree $N = 4$ on Gauss–Lobatto–Legendre integration (GLL) points, which results in a diagonal mass matrix that leads to a simple explicit time integration scheme [16]. As a result, the method lends itself well to calculations on large parallel machines with distributed memory.

We can rewrite the system (2) in matrix form as

$$M\ddot{d} + Kd = F, \tag{3}$$

where $d$ is the displacement vector, $M$ is the diagonal mass matrix, $K$ is the stiffness matrix, $F$ is the source term, and a double dot over a symbol denotes the second derivative with respect to time. For detailed expressions of these matrices, see [16].

The full mesh for the MPI version of our code contains hundreds or thousands of mesh slices in order to mesh the full sphere (Fig. 1, middle). The typical mesh slice extracted from the full mesh to run the serial version of the code is shown in Fig. 1 (left). Each spectral element of the mesh is a deformed hexahedron that contains a topologically-regular but non-evenly spaced grid of points used for numerical integration based on some quadrature rule. In our implementation of the SEM, we use polynomial basis functions of degree $N = 4$; therefore the local grid is composed of five points in each of the three spatial directions. The number of spectral elements in each mesh slice is large, usually between 10,000 and one million. The mesh is 'unstructured' in the finite-element sense, i.e., its topology is not regular and cannot be described by simple $(i, j, k)$ addressing. Therefore, some form of indirect addressing is called for, which implements a non-trivial mapping between the mesh points and the mesh elements. Said differently, but equivalently, the valence of a corner of a cube can be greater than 8, which is the maximum value in a regular mesh of hexahedra. The need for indirect addressing implies that efficient algorithms based on stencils, e.g., finite differences, that can take advantage of cache reuse and optimize prefetch cannot be applied [14,11].

In the SEM algorithm, the time-advancement loop is serial, usually based on an explicit second-order finite-difference time scheme [16]. The time spent in this loop, when running a large number of iterations, typically between $5 \times 10^3$ and $10^5$, dominates the total cost, and is therefore the focus of our efforts. The remainder of the code, which includes preprocessing and postprocessing phases, has negligible cost. Since the mesh is static and the algorithm is explicit, every iteration in the time loop has identical cost in terms of both memory and computation time.

Some of the GLL points are located exactly on the faces, edges or corners of a given spectral element and are shared with its neighbors in the mesh, as illustrated in Fig. 2. Therefore, one can also view the mesh as a set of global grid points with all the multiples, i.e., the common points, counted only once. The number of unique points in each mesh slice is very large, between $5 \times 10^5$ and $10^6$. In the SEM algorithm, mechanical forces computed independently by each element at its $5^3$ points must be summed at the same location of a global vector by all the elements that share a given mesh point. This is called the 'assembly process' in finite-element algorithms, and it implies an atomic sum since different

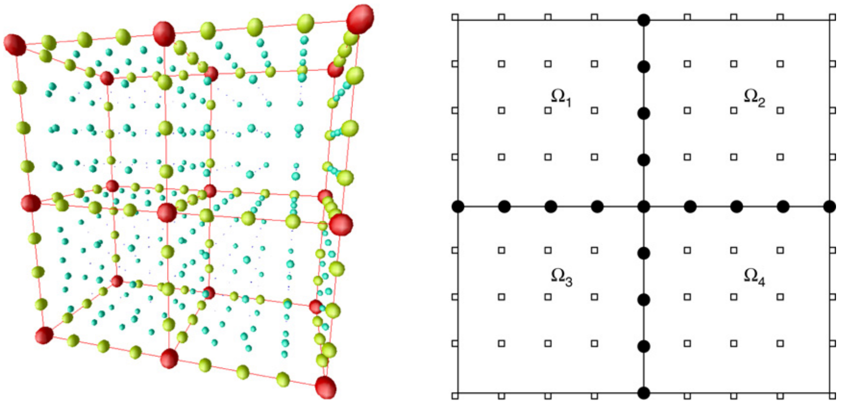

**Fig. 2.** (Left) In a SEM mesh, 3D elements can share points on a face, an edge or a corner; (Right) in 2D, elements can share an edge or a corner. The GLL grid points inside each element are non-evenly spaced but have been drawn evenly-spaced for clarity.

elements add to the same memory location of a global array. In any parallel/threaded code, it is this step that has the greatest impact on performance.

At each iteration of the serial time loop, there are three types of operations performed, which will correspond to our three CUDA kernels in Section 4.1. The SEM algorithm can therefore be summarized as a serial loop over the time steps. At each time step, one executes:

(1) an update with no dependency of some global arrays composed of the unique mesh points
(2) purely local calculations of the product of predefined derivative matrices with a local copy of the displacement vector along cut planes in the three directions ($i, j$ and $k$) of a 3D spectral element, followed by element-by-element products and sums of arrays of dimension $5^3$, which involve a quadruple loop on the dimension of the local arrays, with no dependency between the elements, and finally a sum of these computed values into global arrays composed of unique grid points that can be shared between neighboring spectral elements
(3) an update with no dependency of other global arrays composed of the unique points of the mesh.

## 4. CUDA implementation

In this section, we present two implementations of our code using CUDA. The first runs fully on the GPU, and is therefore expected to generate the fastest speedup, but the problem size is limited by the on-board GPU memory. The host calls a sequence of kernels every time iteration (Fig. 3). This approach was also adopted in [27]. All local and global arrays are stored on the GPU. Pre-processing, i.e., mesh creation, and post-processing are the responsibility of the CPU. Since these are cheap and only done once, their cost is not considered herein. This version of the algorithm is constrained by the memory size of the GPU. On an 8800 GTX card with 768 MB, the maximum problem size occupies 724 MB of memory. We determined this experimentally by executing successive `cudaMalloc()` statements with increments of 1 MB. When the available memory of the CPU exceeds that of the GPU, which is often the case, it is possible to solve larger problems but it then becomes necessary to transfer information to the GPU at every iteration. This data transfer between CPU and GPU decreases the efficiency of the algorithm. We discuss such an approach in the second version in Section 4.2.

In the glossary of Table 1 we briefly explain some of the terms most commonly used in the context of graphics cards and CUDA that are used several times in the rest of the article. For more details the reader is referred to [22].

**Table 1**
Glossary of some of the terms used in CUDA programming and referred to in this article. For more details the reader is referred to [22].

| Term | Explanation |
|---|---|
| Host | The CPU |
| Device | The GPU |
| Kernel | A function executed in parallel on the device |
| Thread block | A set of threads with common access to a shared memory area — inside a block, threads can be synchronized |
| Grid | A set of thread blocks — a kernel is executed on a grid of thread blocks |
| Warp | A group of 16 or 32 threads executed concurrently on a multiprocessor of the GPU |
| Occupancy | The ratio of the actual number of active warps on a multiprocessor to the maximum number of active warps allowed |
| Coalesced memory accesses | Simultaneous GPU global memory accesses coalesced into a single contiguous, aligned memory access |
| *m*-way bank conflicts | Shared memory is divided into memory modules of equal size, called banks, which can be accessed simultaneously. Therefore, any memory read or write request made of *n* addresses that fall in *n* distinct memory banks can be performed simultaneously. A bank conflict occurs whenever two addresses of a memory request fall within the same memory bank, and the access is serialized. A memory request with bank conflicts is split into as many separate conflict-free requests as necessary, decreasing the effective bandwidth by a factor equal to the number *m* separate memory requests. The initial memory request is then said to cause *m*-way bank conflicts. |
| Constant memory | A read-only region of device memory in which reads are accelerated by a cache. |

### 4.1. Implementation 1: Self-contained on the GPU

The code within a CUDA kernel is executed by all the threads on different data (SIMD model). It is launched with several parameters, among which:

- block_size: number of threads per block. Inside a block, threads can share data through high speed shared memory; they can also be synchronized.
- grid_size: number of blocks per grid. The blocks of a grid do not share data and cannot be synchronized.

To carry out synchronization between two sets of blocks, it is necessary to execute them in separate calls to a given kernel or in separate kernels. We express the three steps of the spectral-element algorithm described at the end of Section 3 as three separate kernels, as illustrated in Fig. 3.

The local calculations inside the elements dominate the computation time (kernel 2, see Fig. 4). Each 125-point element is completely independent of the others. We assign one thread

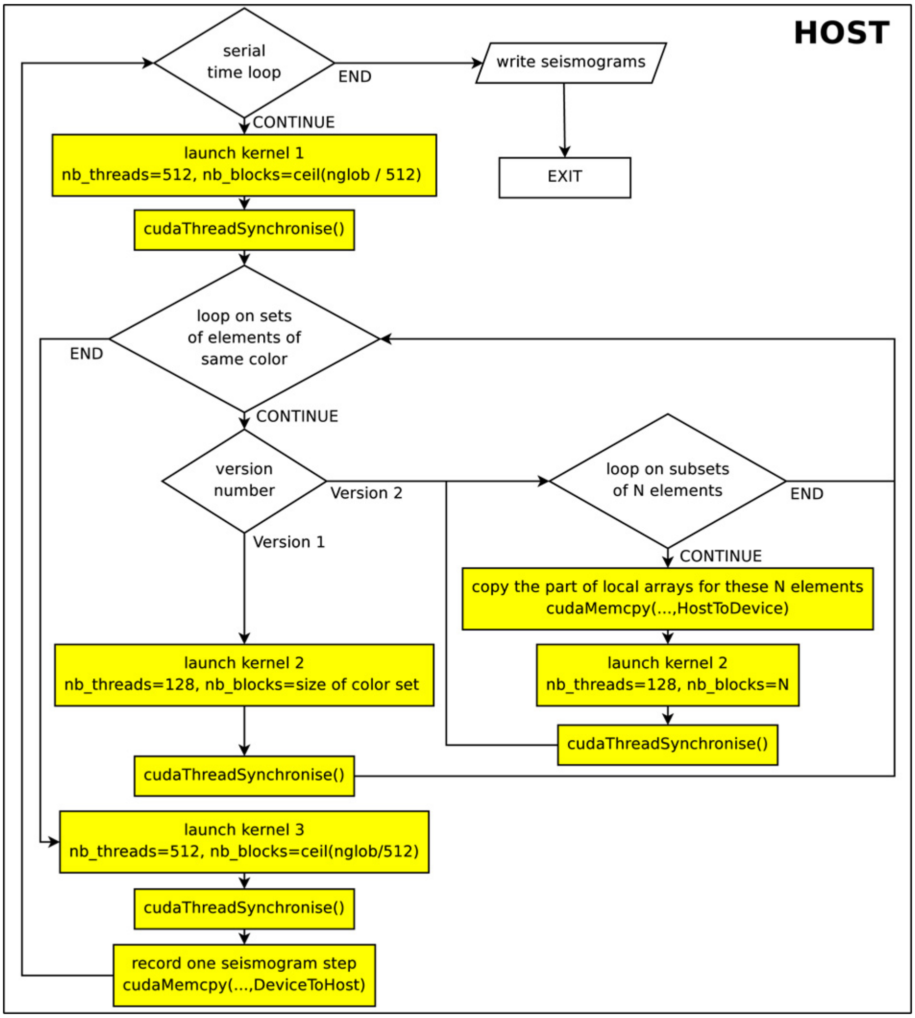
HOST
serial time loop
END
write seismograms
EXIT
CONTINUE
launch kernel 1
nb_threads=512, nb_blocks=ceil(nglob / 512)
cudaThreadSynchronise()
loop on sets of elements of same color
END
CONTINUE
version number
Version 2
Version 1
loop on subsets of N elements
END
CONTINUE
copy the part of local arrays for these N elements
cudaMemcpy(...,HostToDevice)
launch kernel 2
nb_threads=128, nb_blocks=N
cudaThreadSynchronise()
launch kernel 2
nb_threads=128, nb_blocks=size of color set
cudaThreadSynchronise()
launch kernel 3
nb_threads=512, nb_blocks=ceil(nglob/512)
cudaThreadSynchronise()
record one seismogram step
cudaMemcpy(...,DeviceToHost)

**Fig. 3.** Implementation of the main serial time-loop of our spectral-element method on the host for the two versions of our CUDA algorithm. The white boxes are executed on the CPU, while the filled boxes are launched on the GPU.

per point. Thus each CUDA block has 128 threads because using a multiple of the 32-thread warp size is best, and represents a single element.

Benchmarks of the three steps of the serial spectral-element code described at the end of Section 3 indicate that part 2 dominates the computational cost. The fraction of time spent in the various algorithm sections is roughly 6% in part 1, 88% in part 2, and 6% in part 3. Therefore, most optimization efforts should focus on kernel 2, which implements part 2 of the algorithm. Kernels 1 (Fig. 5) and 3, which implement the update of the global arrays, are much simpler and probably already optimal or close to optimal because their occupancy is 100%, memory accesses are perfectly coalesced thanks to choosing blocks of a size that is a multiple of 128 rather than 125, and they are bandwidth-limited since they mostly perform memory accesses and little calculation. Therefore, in what follows, we mostly focus on the analysis of kernel 2. Kernel 3 is similar in structure and properties to kernel 1 and is therefore not presented.

The structure of our spectral-element solver imposes two node numbering systems. Each element has $5^3$ nodes. On the other hand, all the nodes of the mesh are also numbered globally, as explained in Section 3. Each global point of the mesh belongs to one or more spectral elements, depending on whether that mesh point is inside an element and therefore not shared, or on a face, edge or corner, in which case it can be shared. Out of the $5^3 = 125$ points of a given spectral element, $3^3 = 27$ are interior points that cannot be shared, and the 98 others, i.e., the majority may be shared. Each global node can therefore correspond to any number of local nodes depending on the valence of that node. For each mesh point, one must sum the contributions computed independently by each spectral element to a potentially shared global location in the array of global mesh points. We consider two approaches to handle this issue.

#### 4.1.1. Global array update: Mesh coloring

The key challenge in the global update is to ensure that contributions from two local nodes, associated with an identical global node, do not update some global value from different warps. This led us to the concept of coloring, which has successfully been used previously in the context of finite elements on supercomputers [3,12,8] and through which we suppress dependencies between mesh points inside a given kernel. To do this, in our mesh generation preprocessor, which is serial and need not be implemented in CUDA because it is very efficient and run only once, we partition the mesh elements into a finite number of disjoint subsets with the property that any two elements in a given subset do not share any global mesh nodes, as illustrated in Fig. 6. Data at these nodes can therefore be added to their corresponding global location without fear of conflict, removing the need for an atomic mechanism. The local arrays that are related to the mesh elements are reordered based on their color, and an array of offsets that point to the boundary between subsets is created. All elements

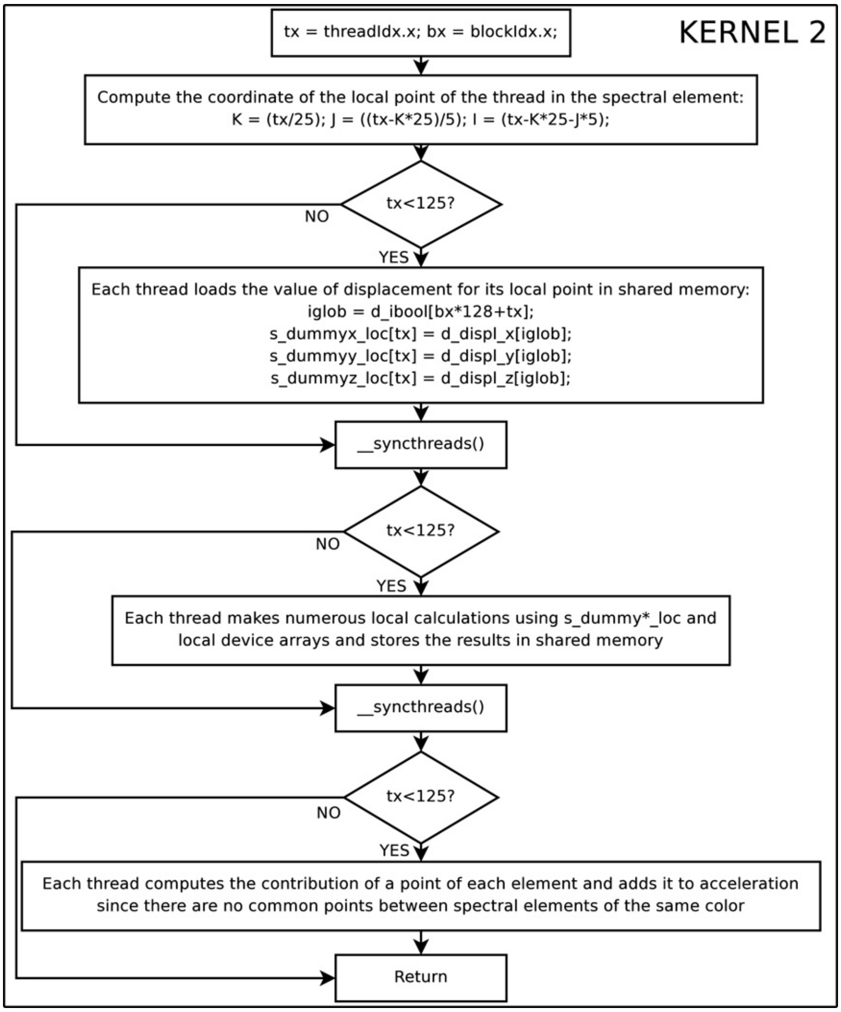
KERNEL 2
tx = threadIdx.x; bx = blockIdx.x;
Compute the coordinate of the local point of the thread in the spectral element:
K = (tx/25); J = ((tx-K*25)/5); I = (tx-K*25-J*5);
tx<125?
NO
YES
Each thread loads the value of displacement for its local point in shared memory:
iglob = d_ibool[bx*128+tx];
s_dummyx_loc[tx] = d_displ_x[iglob];
s_dummyy_loc[tx] = d_displ_y[iglob];
s_dummyz_loc[tx] = d_displ_z[iglob];
__syncthreads()
tx<125?
NO
YES
Each thread makes numerous local calculations using s_dummy*_loc and local device arrays and stores the results in shared memory
__syncthreads()
tx<125?
NO
YES
Each thread computes the contribution of a point of each element and adds it to acceleration since there are no common points between spectral elements of the same color
Return

**Fig. 4.** Flowchart of kernel 2, which is the critical kernel that performs most of the calculations in the spectral-element method and in which we spend 84% of the time.

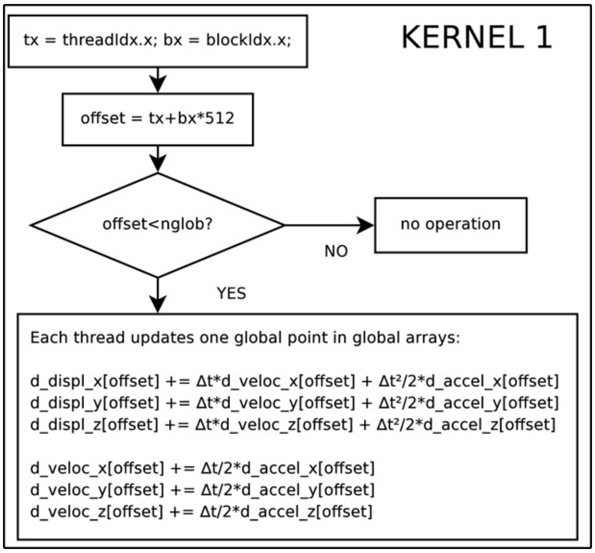
KERNEL 1
tx = threadIdx.x; bx = blockIdx.x;
offset = tx+bx*512
offset<nglob?
NO
no operation
YES
Each thread updates one global point in global arrays:
d_displ_x[offset] += Δt*d_veloc_x[offset] + Δt²/2*d_accel_x[offset]
d_displ_y[offset] += Δt*d_veloc_y[offset] + Δt²/2*d_accel_y[offset]
d_displ_z[offset] += Δt*d_veloc_z[offset] + Δt²/2*d_accel_z[offset]
d_veloc_x[offset] += Δt/2*d_accel_x[offset]
d_veloc_y[offset] += Δt/2*d_accel_y[offset]
d_veloc_z[offset] += Δt/2*d_accel_z[offset]

**Fig. 5.** Flowchart of kernel 1, which is a far less critical kernel that performs the update of some global arrays in the spectral-element method and in which we only spend 7% of the time. It is probably optimal or close to optimal because its occupancy is 100% and all its memory accesses are perfectly coalesced.

are processed by calling the kernel repeatedly with a grid size equal to the number of elements in a particular color subset.

Our basic algorithm to color the mesh is the following:

```
 while (all elements are not colored) {
   change current color
     for each element {
       if (this element is not colored) {
       if (its neighbors do not have current color)
         { color this element with current color }
} } }
```

While this algorithm is simple, it generates color distributions that are very uneven. The initial colors found contain far more elements than the last. We therefore add a second step that balances the number of elements inside each mesh subset of a given color to finally obtain subsets of comparable size. This second step is as follows: we have "rich" sets that contain too many elements and "poor" sets that contain too few. Richest colors can give elements to poorest colors after checking that these elements do not share a point with the set they are added to. The richest set gives elements to poor sets, starting with the poorest. This second step stops either when most sets contain a number of elements that is close to the mean value (the total number of elements divided by the number of sets), which is the target, or when rich sets cannot give elements to poor sets any longer because these elements would otherwise share a point with other elements in these poor sets.

Were this not done, CUDA kernels running on the last mesh subsets, which might contain only a few elements, would become

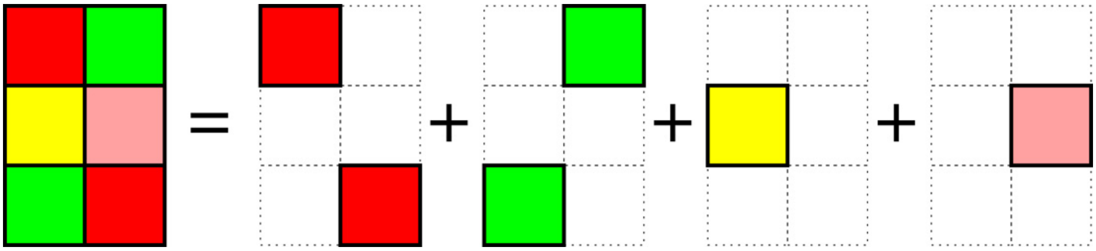

**Fig. 6.** Illustration of coloring: A connected mesh can always be decomposed into subsets of disjoint mesh elements only, suppressing the need for an atomic update.

inefficient because of insufficient active blocks. One can concoct small examples that demonstrate that this simple algorithm is not optimal: the number of colors obtained is not always the minimum possible on a given mesh. Nonetheless, it is sufficient in practice.

### 4.2. Implementation 2: Shared CPU/GPU

In many current PCs, the memory installed on the CPU motherboard is often four to eight times larger than the amount installed on the graphics card. It is the case in our experimental setup: 0.75 GB on our 8800 GTX GPU, 1 GB on our GTX 280 GPU, and 8 GB on each PC. This implies that the size of a problem that runs 100% on the GPU is necessarily smaller than if it were run on the CPU. It is therefore of interest to implement another CUDA version that is not too drastically restricted by the amount of device memory of the graphics card.

In this second version (Fig. 3), we store all the global arrays on the GPU and use the remaining space to store the local arrays of some maximum number of elements, with the constraint that all arrays loaded correspond to a fixed color. Processing of these elements is then exactly the same as in the first version of the algorithm. The cost incurred during the transfer of the local arrays to the GPU decreases the efficiency of the algorithm, but larger problems can now be solved.

Given a partition of the spectral elements into subsets of fixed color, these subsets are further divided into packets. The size of these packets is precisely the grid size of kernel 2, i.e., the number of thread blocks sent to the GPU. This version is thus limited by the space taken by the global arrays on the GPU. At worst, space must be left on the GPU for a single element, although for a packet size of one, the algorithm is expected to be extremely inefficient for lack of parallelism. Thus, we aim to maximize the size of the packets, although this can only be done at the expense of space taken up by the global arrays. The global arrays account for approximately 30% of the memory size of the spectral-element code. As a consequence, this out-of-GPU-core version can handle problems with three times the number of elements in the version running fully on the GPU.

One easily computes an estimate of the maximum model size that can be run with this second version. Assuming $\texttt{nspec}$ elements and $\texttt{nglob} \simeq (\texttt{nspec} \times 125)/2$ global points, because in a typical mesh many points are shared and about half are duplicates that are removed when creating the global mesh of unique points, the space taken by the 9 global (SGA) and 11 local (SLA) arrays is (in bytes)

$$\text{SGA} \simeq 9 \times \texttt{nglob} \times \text{sizeof(float)}$$

$$\text{SLA} = 11 \times \texttt{grid\_size} \times 128 \times \text{sizeof(float)}.$$

For SLA, each element has a size of 128 instead of 125 due to padding used to align memory accesses, as explained in Section 5. Therefore the total size of the arrays on the card is SC = SGA + SLA. Setting SC equal to the amount of user-controlled memory on the graphics card, and expressing $\texttt{nglob}$ as a function of the number of elements, we find that, knowing that a float takes 4 bytes, and for a card with 724 MB usable memory,

$$\texttt{nspec} = (\text{SC} - \text{SLA})/2250 = \frac{724 \times 2^{20} - 5632 \times \texttt{grid\_size}}{2250}.$$

The maximum problem size ranges from $\texttt{nspec} = 332{,}282$ for packets of $\texttt{grid\_size} = 2048$ elements to $\texttt{nspec} = 316{,}903$ for packets of $\texttt{grid\_size} = 8192$. Therefore the maximum size of the problem does not vary significantly with the size of the packets. We will see below in Fig. 8 that to keep a good performance level one should not select packets smaller than 1024 elements. A problem with 320,000 elements requires approximately 2.4 GB on the CPU, or 3.3 times the size of usable GPU memory. Therefore, this version can handle problems more than three times the size of the first version.

## 5. Optimizations common to the two CUDA implementations

In this section, we discuss some of the optimizations incorporated into the kernels. As a rule-of-thumb, the fastest kernels minimize access to device memory, avoid non-coalesced accesses to global memory, avoid bank conflicts when reading from or writing to shared memory, and try to minimize register and/or shared memory usage to maximize occupancy. At the same time, one strives to work with many blocks running per multiprocessor to overlap the latencies of memory transfers with useful computation.

Accesses to the global memory of the GPU are not cached, and thus have a huge latency in the range of 400 to 600 cycles. The thread scheduler is in charge of hiding this latency by executing other thread blocks on the multiprocessors while waiting for memory and other requests to be serviced. In a CUDA kernel, it is thus important to launch a sufficiently large number of blocks so that each multiprocessor has many blocks executing at any given time.

To ensure coalesced reads from global memory, the local array sizes are a multiple of 128 floats, which is itself a multiple of the half-warp size of 16, instead of $5^3 = 125$, thus purposely sacrificing $128/125 = 1.023 = 2.3\%$ of memory. Each thread is responsible for a different point in the element. Consequently, the threads of a half-warp load adjacent elements of a float array. Access to global memory is thus perfectly coalesced in kernels 1 and 3, as well as in the parts of kernel 2 that access local arrays. When accessing global arrays in kernel 2, the indirect addressing necessary to handle the unstructured mesh topology results in non-coalescent accesses, as seen in Section 3.

The 5 × 5 derivative matrices are stored in constant memory, which has faster access times and a cache mechanism. All threads of a half-warp can access the same constant in one cycle.

Each multiprocessor of the NVIDIA 8800 GTX has 8192 registers and 16 kbytes of shared memory in each multiprocessor. Each block uses the same amount of shared memory. As the shared memory used per block increases, fewer blocks can run concurrently, and therefore, fewer threads are active. With 128 threads per block, a maximum of 6 blocks, i.e., 768 threads per processor divided by 128 threads per block can run concurrently. This implies that all threads are occupied if the kernel uses 8192/768 $\simeq$ 10 registers or fewer per thread because to get 100% occupancy one needs all threads active. The key to lowering register usage is to either launch multiple kernels with a less complex structure, or define variables local to the kernel, which are stored in registers, as close as possible to where they are used. Unfortunately, the dense matrices and equations we are solving make this separation very difficult if not impossible. As a result, 27 registers per thread is the lowest number we have achieved, using 6160 bytes of shared memory with a corresponding occupancy of 33%. To minimize register usage, we have availed ourselves of some tricks: use of the "volatile" keyword, and use of the -maxrregcount $= N$ compilation flag that limits the number of registers used. Unfortunately, the latter approach leads to storage of variables in local memory, i.e., device memory, with a resulting decrease in performance. The remaining two kernels have simpler structure and attain 100% occupancy with coalesced read/writes.

Next, we determined whether kernel 2 was bandwidth- or compute-bound. To do this, we removed all calculations from kernel 2 and only read every local array once and wrote a simple arithmetic expression making use of all the input data to prevent the CUDA compiler from suppressing them. For example, if there were three input arrays and one output array, the resulting kernel would be `read a; read b; read c; write d = a + b + c;`. This artificial kernel has occupancy of 100% and is bandwidth-dominated. Under this extremely favorable condition of perfect occupancy, the performance is only 1.6 times better than kernel 2. We might have expected a higher gain, typically around a factor of 3, given the factor of 3 improvement in occupancy. This clearly indicates that kernel 2 is bandwidth-bound: performance is limited by the memory bandwidth of the card and under the best of circumstances cannot increase by more than a factor of 1.6, regardless of how it is coded. Therefore, we know that the 15x speedup that we measure for our application on the 8800 GTX is very good because even with perfect occupancy we could only increase it to roughly 15 $\times(1.6 \times 84\%) \simeq 20$ since kernel 2 is the critical kernel that takes 84% of the time, as mentioned above, and the other kernels already have an occupancy of 100%.

Arrays that store variables in a local element are three-dimensional, and access can occur along any of the three dimensions. We have 5-way bank conflicts (see the glossary of Table 1 for a definition of bank conflict) in the section of kernel 2 (see Fig. 4) that performs local calculations inside each spectral element with loops on its $5 \times 5 \times 5$ local points. This happens because at some point five threads access the same memory bank $L$, five other threads access memory bank $L+5$, and so on. We have not found a satisfactory way of reducing or suppressing these bank conflicts. The issue could perhaps be resolved at the expense of additional memory consumption, but on memory-limited graphics cards this is not practical. It is quite possible that such bank conflicts simply cannot be suppressed in such a high-order finite-element application.

We also tried to use fast operations on the card but did not notice any measurable improvement: a kernel in which all standard operations were replaced by fast operations: `__fmul_rz()`, `__fadd_rz()` for floats and `__mul24()` for integers did not measurably improve performance.

## 6. Numerical validation of the two CUDA versions

In Fig. 7 we validate the two CUDA implementations of our spectral-element code by comparing the time evolution of the vertical component of the displacement vector recorded at one grid point, called a 'seismogram' in the field of seismic wave propagation, produced by waves generated by an earthquake source located at another grid point and propagating across the mesh. We take the mesh of Fig. 1 (left) and put the earthquake source in element 3711 at global grid point 256,406 and record the vertical component of the displacement vector in element 7413 at global grid point 495,821. As a comparison, we run the same simulation with the original C-code running fully on the CPU using either single precision or double precision floats. The four seismograms are indistinguishable at the scale of the figure, which validates our two CUDA implementations and also validates the fact that single-precision arithmetic is adequate for this problem. The results differ in the very last decimals (only) because of the different order in which the operations are performed, which produces a different roundoff as shown by the difference amplified by 10,000 between CUDA version 1 and the single-precision serial C version.

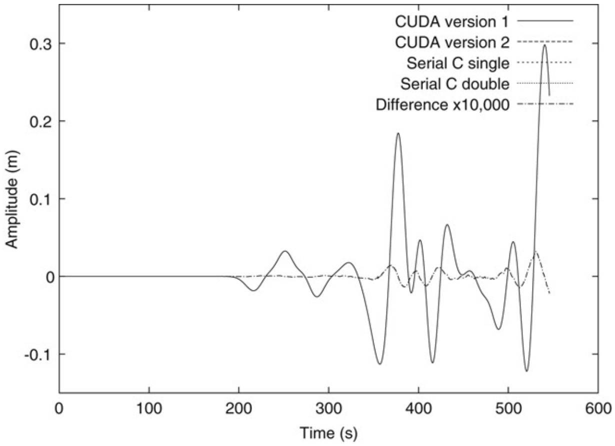

**Fig. 7.** Vertical component of the displacement vector recorded at one grid point produced by waves generated by an earthquake source located at another grid point and propagating across the mesh of Fig. 1 computed with our two CUDA implementations of the code as well as with the original serial version of the C-code using either single-precision or double-precision floats. The difference amplified by 10,000 between CUDA version 1 and the single-precision serial C version is very small and validates our implementation. The other seismograms are in excellent agreement as well, in particular the single-precision and double-precision serial codes, which shows that single precision is sufficient for this problem.

## 7. Performance analysis and speedup obtained

Our experimental setup is composed of an NVIDIA GeForce 8800 GTX card installed on the PCI Express 1 bus of a dual-processor dual-core 64 bit Intel Xeon E5345 2.33 GHz PC with 8 GB of RAM and running Linux kernel 2.6.23; and of an NVIDIA GeForce GTX 280 card installed on the PCI Express 1 bus of a dual-processor dual-core 64 bit AMD Opteron PC with 8 GB of RAM and running Linux kernel 2.6.20. The 8800 GTX card has 16 multiprocessors, i.e., 128 cores, and 768 MB of memory, and the memory bandwidth is 86.4 GB per second with a memory bus width of 384 bits. The GTX 280 card has 240 cores and 1024 MB of memory, and the memory bandwidth is 141.7 GB per second with a memory bus width of 512 bits. We use CUDA version 2 beta, driver 169.09, and the following three compilers with compilation options:

| | |
|---|---|
| icc version 10.1: | -no-ftz |
| gcc version 4.1.2: | -fno-trapping-math |
| nvcc version CUDA_v2_beta: | -fno-trapping-math. |

Floating-point trapping is turned off because underflow-trapping occurs very often in the initial time steps of many seismic propagation algorithms and can lead to severe slowdown.

The serial code we start from is already heavily optimized [17, 15], in particular using the ParaVer performance analysis tool [13]

**Table 2**
Time per element and speedup obtained on the 8800 GTX for version 1 of our CUDA algorithm (entirely on the GPU) and version 2 (shared CPU/GPU) using an increasingly larger problem size. For version 2 we also show the percentage of time taken by transfers, which is very high because the spectral-element algorithm is bandwidth-bound.

| Mesh size (MB) | Version 1 | | Version 2 | | |
|---|---|---|---|---|---|
| | Time/elt (µs) | Speedup | Time/elt (µs) | Speedup | Transfer time (%) |
| 65 | 1.5 | 13.5 | 4.2 | 4.6 | 68 |
| 405 | 1.3 | 15 | 3.7 | 5.3 | 68 |
| 633 | 1.3 | 15 | 3.7 | 5.3 | 67 |

**Table 3**
Same as Table 2 but measured on the GTX 280.

| Mesh size (MB) | Version 1 | | Version 2 | | |
|---|---|---|---|---|---|
| | Time/elt µs | Speedup | Time/elt µs | Speedup | Transfer time (%) |
| 65 | 0.91 | 21.5 | 4.1 | 4.8 | 80 |
| 405 | 0.8 | 24.8 | 2.93 | 6.8 | 76 |
| 633 | 0.78 | 25.3 | 2.8 | 7.1 | 75 |

to minimize cache misses, therefore high speedups will be difficult to reach because our serial reference code is very fast. It is based on a parallel version that won the Gordon Bell supercomputing award on the Japanese Earth Simulator – a NEC SX machine – at the SuperComputing'2003 conference [17]. To illustrate this, let us study the influence of the compiler on CPU time of the serial version of the code, measuring total time for 2100 time iterations for increased reliability:

```
icc -O1: 324.6 s ; -O2: 345.0 s ; -O3: 345.2 s
gcc -O1: 336.8 s ; -O2: 313.7 s ; -O3: 302.2 s
```

The fact that differences are small and that for `icc` a less aggressive optimization level is more efficient underlines the fact that the code is already well optimized. We select `gcc -O3` as a reference for all the timing measurements of the serial code presented below. The small differences in serial performance will not significantly affect our conclusions.

To make sure that converting the original code from Fortran95 to C did not significantly change performance levels, we also timed a serial version of the original Fortran95 code, run on the same mesh and therefore same problem size, with both Intel ifort version 9.1 and GNU gfortran version 4.1.2, using `-O1`, `-O2` and `-O3`. In the best case, we obtained:

```
gfortran -O3: 328.7 s
ifort -O2:  270.1 s
```

i.e., the difference between the best C time and best Fortran95 time is only $302.2/270.1 = 1.12 = 12\%$.

Let us now study the speedup obtained for increasingly larger i.e. higher-resolution versions of the mesh in Fig. 1. We consider three mesh sizes: a low-resolution mesh of 65 MB of memory, a medium-resolution mesh of 405 MB, and a high-resolution mesh of 633 MB. The three meshes contain respectively (7424, 496,525), (46,592, 3061,529), and (72,800, 4769,577) elements and global points. Tables 2 and 3 show the speedup measured for version 1 of the code (entirely on the GPU) and version 2 (shared CPU/GPU) on both cards. Values vary weakly with the size of the mesh. The maximum speedup is 25x on the GTX 280 and 15x on the 8800 GTX for version 1, and 5.3x on the 8800 GTX and 7.1x on the GTX 280 for version 2, which is significantly slower but can handle problems more than three times larger than the memory size of the GPU.

Fig. 8 shows the computation time per mesh element in kernel 2, which is by far the most computation-intensive kernel (excluding host/device transfer times; only computation time is measured), measured on the 8800 GTX as a function of the number of mesh elements in each packet of this second version of our algorithm. Each spectral element of our mesh, which contains 125 points, leads to a block of 128 threads. One can therefore consider this figure as a graph of compute time as a function of `grid_size` because the number of elements is equal to the number of blocks. One clearly observes that variations of performance are relatively small, i.e., computation time per element asymptotes, when packets of reasonably large size are used but that performance decreases very significantly if the packets are too small, typically below 512 elements, because the scheduler does not have enough blocks to overlap latencies. High-frequency variations can also be observed owing to the fact that we made measurements with a unit increment for the number of mesh elements in each packet.

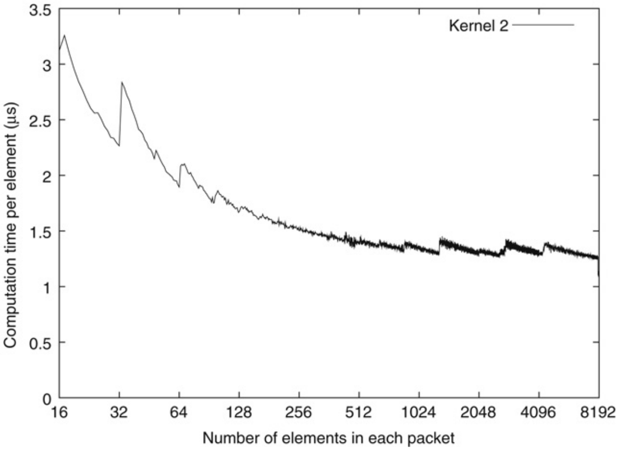

**Fig. 8.** Evolution of computation time per mesh element in kernel 2 as a function of the number of mesh elements in each packet of the second version of our algorithm.

## 8. Lessons learned, conclusions and future work

We have ported a high-order spectral-element application, which performs the numerical simulation of seismic wave propagation resulting from earthquakes, on NVIDIA GeForce 8800 GTX and GTX 280 graphics cards using CUDA. Since this application runs in single precision, current GPU hardware that supports quasi-IEEE-754 single precision arithmetic is suitable and sufficient.

We discussed two possible implementations of the code: the first is limited to the memory size of the card, and the second can handle larger cases because it is only limited by the size of the global arrays, because often the amount of memory installed on the CPU side is significantly larger than the amount of memory on the graphics device. We validated the two algorithms by comparison to the results of the same run with the original C version of the code without CUDA. We then performed several numerical tests to compare the performance between the two versions and with respect to the original version without CUDA and showed that in the best case we obtained a performance increase of 25x.

In future work, we would like to investigate the use of several cards on the same PC, the so-called multi-GPU setup. In addition, coupling with MPI could further accelerate our code. The key issues will be to minimize the serial components of the code to avoid the effects of Amdahl's law and to overlap MPI communications with calculations. The combination of MPI and GPUs would allow us to run large-scale realistic cases similar to those in [17,15], but at a vastly reduced cost.

## Acknowledgments

The authors would like to thank Christophe Merlet, Jean Roman, Cyril Zeller and Bruno Jobard for fruitful discussions. The Web page of Dominik Göddeke on GPGPU and the http://www.gpgpu.org Web site were very useful. The comments of three anonymous reviewers improved the manuscript. This research was funded in part by French ANR grant NUMASIS ANR-05-CIGC-002 and by US National Science Foundation grant NSF-0345656. The authors thank Hélène Barucq and INRIA for funding a one-month visit of Gordon Erlebacher to University of Pau, during which a significant part of this work was performed.

## References

[1] R. Abdelkhalek, Évaluation des accélérateurs de calcul GPGPU pour la modélisation sismique, Master's Thesis, ENSEIRB, Bordeaux, France, 2007.

[2] J.A. Anderson, C.D. Lorenz, A. Travesset, General purpose molecular dynamics simulations fully implemented on graphics processing units, J. Comput. Phys. 227 (10) (2008) 5342–5359.

[3] P. Berger, P. Brouaye, J.C. Syre, A mesh coloring method for efficient MIMD processing in finite element problems, in: Proceedings of the International Conference on Parallel Processing, ICPP'82, August 24–27, 1982, Bellaire, Michigan, USA, IEEE Computer Society, 1982, pp. 41–46.

[4] T. Brandvik, G. Pullan, Acceleration of a two-dimensional Euler flow solver using commodity graphics hardware, in: Proceedings of the Institute of Mechanical Engineers, Part C: J. Mech. Eng., Part C: J. Mech. Eng. Sci. 221 (12) (2007) 1745–1748.

[5] I. Buck, GeForce 8800 and NVIDIA CUDA: A new architecture for computing on the GPU, in: Proceedings of the Supercomputing'06 Workshop on "General-Purpose GPU Computing: Practice and Experience", 2006. URL www.gpgpu.org/sc2006/workshop/presentations/Buck_NVIDIA_Cuda.pdf.

[6] D. Dobb's, Dr. Dobb's Portal web site (March 2008). URL www.ddj.com/hpc-high-performance-computing/207200659.

[7] R. Dolbeau, S. Bihan, F. Bodin, HMPP: A hybrid multi-core parallel programming environment, in: Proceedings of the Workshop on General Purpose Processing on Graphics Processing Units, GPGPU'2007, Boston, MA, USA, 2007. URL www.irisa.fr/caps/projects/Astex.

[8] C. Farhat, L. Crivelli, A general approach to nonlinear finite-element computations on shared-memory multiprocessors, Comput. Methods Appl. Mech. Engrg. 72 (2) (1989) 153–171.

[9] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S.H.M. Buijssen, M. Grajewski, S. Turek, Exploring weak scalability for FEM calculations on a GPU-enhanced cluster, Parallel Comput. 33 (10–11) (2007) 685–699.

[10] D. Göddeke, R. Strzodka, S. Turek, Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations, Internat. J. Parallel Emerg. Distrib. Syst. 22 (4) (2007) 221–256.

[11] N.K. Govindaraju, D. Manocha, Cache-efficient numerical algorithms using graphics hardware, Parallel Comput. 33 (2007) 663–684.

[12] T.J.R. Hughes, R.M. Ferencz, J.O. Hallquist, Large-scale vectorized implicit calculations in solid mechanics on a Cray X-MP/48 utilizing EBE preconditioned conjugate gradients, Comput. Methods Appl. Mech. Engrg. 61 (2) (1987) 215–248.

[13] G. Jost, H. Jin, J. Labarta, J. Giménez, J. Caubet, Performance analysis of multilevel parallel applications on shared memory architectures, in: Proceedings of the IPDPS'2003 International Parallel and Distributed Processing Symposium, Nice, France, 2003. URL www.cepba.upc.es/paraver.

[14] T. Kim, Hardware-aware analysis and optimization of 'Stable Fluids', in: Proceedings of the ACM Symposium on Interactive 3D Graphics and Games, 2008.

[15] D. Komatitsch, J. Labarta, D. Michéa, A simulation of seismic wave propagation at high resolution in the inner core of the Earth on 2166 processors of MareNostrum, in: Lecture Notes in Computer Science, vol. 5336, 2008, pp. 364–377.

[16] D. Komatitsch, J. Tromp, Introduction to the spectral-element method for 3-D seismic wave propagation, Geophys. J. Int. 139 (3) (1999) 806–822. URL www.geodynamics.org/cig/software/packages/seismo.

[17] D. Komatitsch, S. Tsuboi, C. Ji, J. Tromp, A 14.6 billion degrees of freedom, 5 teraflops, 2.5 terabyte earthquake simulation on the Earth Simulator, in: Proceedings of the ACM/IEEE Supercomputing SC'2003 Conference, 2003, pp. 4–11.

[18] K. Liu, X.B. Wang, Y. Zhang, C. Liao, Acceleration of time-domain finite element method (TD-FEM) using Graphics Processor Units (GPU), in: Proceedings of the 7th International Symposium on Antennas, Propagation & EM Theory, ISAPE '06, Guilin, China, 2006.

[19] Q. Liu, J. Polet, D. Komatitsch, J. Tromp, Spectral-element moment-tensor inversions for earthquakes in Southern California, Bull. Seismol. Soc. Amer. 94 (5) (2004) 1748–1761.

[20] P. McCormick, J. Inman, J. Ahrens, J. Mohd-Yusof, G. Roth, S. Cummins, Scout: A data-parallel programming language for graphics processors, Parallel Comput. 33 (2007) 648–662.

[21] P. Messmer, P.J. Mullowney, B.E. Granger, GPULib: GPU computing in high-level languages, Comput. Sci. Engrg. 10 (5) (2008) 70–73.

[22] NVIDIA, CUDA (Compute Unified Device Architecture) Programming Guide Version 1.1, NVIDIA Corporation, Santa Clara, CA, USA, 143 pages (November 2007).

[23] NVIDIA, NVIDIA GeForce GTX 200 GPU architectural overview, second-generation unified GPU architecture for visual computing, Tech. Rep., NVIDIA, 2008. URL www.nvidia.com/docs/IO/55506/GeForce_GTX_200_GPU_Technical_Brief.pdf.

[24] L. Nyland, M. Harris, J. Prins, Fast N-body simulation with CUDA, in: GPU Gems 3, Addison-Wesley Professional, 2007, pp. 677–695 (Chapter 31).

[25] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn, T.J. Purcell, A survey of general-purpose computation on graphics hardware, Comput. Graph. Forum 26 (1) (2007) 80–113.

[26] E. Petit, F. Bodin, Extracting threads using traces for system on a chip, in: Proceedings of the Compilers for Parallel Computers, CPC'2006, La Coruña, Spain, 2006.

[27] Z.A. Taylor, M. Cheng, S. Ourselin, High-speed nonlinear finite element analysis for surgical simulation using Graphics Processing Units, IEEE Trans. Med. Imaging 27 (5) (2008) 650–663.

[28] M. Woolsey, W.E. Hutchcraft, R.K. Gordon, High-level programming of graphics hardware to increase performance of electromagnetics simulation, in: Proceedings of the 2007 IEEE International Symposium on Antennas and Propagation, 2007.

[29] W. Wu, P.A. Heng, A hybrid condensed finite element model with GPU acceleration for interactive 3D soft tissue cutting: Research articles, Comput. Animat. Virtual Worlds Archive. 15 (3–4) (2004) 219–227.

[30] W. Wu, P.A. Heng, An improved scheme of an interactive finite element model for 3D soft-tissue cutting and deformation, Vis. Comput. 21 (8–10) (2005) 707–717.

**Dimitri Komatitsch** is a Professor of Computational Geophysics at University of Pau, France. He was born in 1970 and did his Ph.D. at Institut de Physique du Globe de Paris, France, in 1997.

**David Michéa** is a researcher at INRIA, University of Pau and CNRS, France. He was born in 1973 and did his Master's thesis at University of Strasbourg, France, in 2006.

**Gordon Erlebacher** is a Professor of Computer Science at Florida State University, Tallahassee, USA. He was born in 1957 and did his Ph.D. at Columbia University in New York, USA in 1983.