High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster

Dimitri Komatitsch[a,b,1], Gordon Erlebacher[c], Dominik Göddeke[d], David Michéa[a,2]

[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*
[d] *Institut für Angewandte Mathematik, TU Dortmund, Germany*

Abstract

We implement a high-order finite-element application, which performs the numerical simulation of seismic wave propagation resulting for instance from earthquakes at the scale of a continent or from active seismic acquisition experiments in the oil industry, on a large cluster of NVIDIA Tesla graphics cards using the CUDA programming environment and non-blocking message passing based on MPI. Contrary to many finite-element implementations, ours is implemented successfully in single precision, maximizing the performance of current generation GPUs. We discuss the implementation and optimization of the code and compare it to an existing very optimized implementation in C language and MPI on a classical cluster of CPU nodes. We use mesh coloring to efficiently handle summation operations over degrees of freedom on an unstructured mesh, and non-blocking MPI messages in order to overlap the communications across the network and the data transfer to and from the device via PCIe with calculations on the GPU. We perform a number of numerical tests to validate the single-precision CUDA and MPI implementation and assess its accuracy. We then analyze performance measurements and depending on how the problem is mapped to the reference CPU cluster, we obtain a speedup of 20x or 12x.

Key words: GPU computing, finite elements, spectral elements, seismic modeling, CUDA, MPI, speedup, cluster.
PACS:

1. Introduction

Over the past several years, the number of non-graphical applications ported to graphics processing units (GPUs) has grown at an increasingly fast pace, with a commensurate shift from the early challenges of implementing simple algorithms to more demanding and realistic application domains. The research field of GPGPU ('general purpose computation on GPUs') or 'GPU computing' has clearly been established, see for instance recent surveys on the topic [1, 2, 3, 4]. Among the reasons that explain why GPU computing is picking up momentum, the most important is that the use of GPUs often leads to speedup factors between 5x and 50x on a single GPU versus a single CPU core, depending on the application. GPU peak performance continues to improve at a rate substantially higher than CPU performance. The new Fermi architecture [5] is one more step in this direction.

1.1. GPU computing

In 1999 researchers began to explore the use of graphics hardware for non-graphics operations and applications. In these early years, programming GPUs for such tasks was cumbersome because general

*(1) Corresponding author.
**(2) Now at: Bureau de Recherches Géologiques et Minières, 3 avenue Claude Guillemin, BP 36009, 45060 Orléans Cedex 2, France.
Email addresses: dimitri.komatitsch@univ-pau.fr (Dimitri Komatitsch), gerlebacher@fsu.edu (Gordon Erlebacher), dominik.goeddeke@math.tu-dortmund.de (Dominik Göddeke), davidmichea@gmail.com (David Michéa)
URL: http://www.univ-pau.fr/~dkomati1 (Dimitri Komatitsch), http://www.sc.fsu.edu/~erlebach (Gordon Erlebacher), http://www.mathematik.tu-dortmund.de/~goeddeke (Dominik Göddeke)

operations had to be cast into graphics constructs. Nonetheless, the field of GPGPU emerged slowly, with more and more researchers conducting a number of ground-breaking research work. We refer to the PhD thesis by one of the authors [6] and a survey article by Owens et al. [7] for a comprehensive overview. The situation changed dramatically in 2006, when NVIDIA released the initial version of CUDA. It constitutes both a hardware architecture for throughput-oriented computing and an associated programming model [8, 9, 10, 4]. CUDA gives users a significant amount of control over a powerful streaming/SIMD manycore architecture. It is being used to study a diverse set of compute-intensive physical problems, including fluid dynamics, computational chemistry and molecular dynamics, medical imaging, astrophysics and geophysics, cf. Section 2 for an overview of related work in the context of this article. Recently, OpenCL has been released as an open industry standard to facilitate portability and vendor-independence, targeting both GPUs and multicore CPUs [11, 4].

The main reason why GPUs outperform CPUs significantly on many workloads is that the architecture design favors the throughput of many data-parallel tasks over the latency of single threads. This performance results chiefly through massive hardware multithreading to cover latencies of off-chip memory accesses. In that regard, Fatahalian and Houston [12] provide a comparison of the architecture of some of the latest GPUs, as well as potential future GPU and modern multicore processor designs.

1.2. The spectral-element method and SPECFEM3D, our seismic wave propagation application

In the last decade, together with several colleagues, we developed SPECFEM3D, a software package that performs the three-dimensional numerical simulation of seismic wave propagation resulting from earthquakes in the Earth or from active seismic experiments in the oil industry, based on the spectral-element method (SEM) [13, 14, 15, 16]. The SEM is similar to a high-order finite-element method with high-degree polynomial basis functions. Based on the amount of published literature, it is likely the second most used numerical technique to model seismic wave propagation in complex three-dimensional models, after the finite-difference method.

The spectral-element method is extremely efficient to model seismic wave propagation because it is designed to combine the good accuracy properties of pseudospectral techniques such as Legendre or Chebyshev methods with the geometrical flexibility of classical low-order finite-element methods. In that regard, it can be seen as a close-to-optimal *hp* method obtained by combining the advantages of h methods that use a dense mesh with the advantages of p methods that use high-degree basis functions (see for instance [17, 18, 19, 20, 21, 22, 23, 24], among many others). The use of Gauss-Lobatto-Legendre points (instead of Gauss) leads to a diagonal mass matrix and therefore to a fully explicit algorithm, see Section 3. This in turn leads to a very efficient implementation, in particular on parallel computers (see for instance [13, 25, 26, 24]).

However, an important limitation of the classical SEM is that one needs to design a mesh of hexahedra, which in the case of very complex geological models can be a difficult and time consuming process. If needed, one can turn to more complex SEM implementations with a mixture of hexahedra, tetrahedra and pyramids (see for instance [27, 28, 29, 30, 24]), thus making mesh generation easier in particular in the case of very complex models or geometries. Another option in such a case is to turn to discontinuous Galerkin methods [31, 32, 33, 34, 35, 36, 37], which can also combine different types of mesh elements.

1.3. Article contribution

In order to study seismic wave propagation in the Earth at very high resolution (i.e., up to very high seismic frequencies) the number of mesh elements required is very large. Typical runs require a few hundred processors and a few hours of elapsed wall-clock time. Large simulations run on a few thousand processors, typically 2000 to 4000, and take two to five days of elapsed wall-clock time to complete [13, 38]. The largest calculation that we have performed ran on close to 150,000 processor cores with a sustained performance level of 0.20 petaflops [25].

In this article, we propose to extend SPECFEM3D to a cluster of GPUs to further speed up calculations by more than an order of magnitude, or alternatively, to perform much longer physical simulations at the same cost. The two key issues to address are 1) the minimization of the serial components of the code to avoid the effects of Amdahl's law and 2) the overlap of MPI communications with calculations.

In finite-element applications, whether low or high order, contributions must be added between adjacent elements, which leads to dependencies for degrees of freedom that require the summation of contributions from several elements. This so-called 'assembling' process is the most difficult component of the SEM algorithm to handle in terms of obtaining an efficient implementation. Several strategies can be used to implement it [24]. In a parallel application, atomic sums can be used to handle these dependencies. Unfortunately, in general atomic operations can lead to reduced efficiency, and in addition on current NVIDIA GPUs atomic operations can only be applied to integers. We will therefore use mesh coloring instead to

define independent sets of mesh elements, ensuring total parallelism within each set. We will show that coloring, combined with non-blocking MPI to overlap communications between nodes with calculations on the GPU, leads to a speedup of 20x or 12x, depending on how speedup is defined. It is important to point out that most of the currently published speedups are on a single GPU, while in this study we maintain the same performance level even on a large cluster of GPUs.

The outline of the article is as follows: in Section 2 we describe similar problems ported to the GPU by others. In Section 3 we discuss the serial algorithm implemented in SPECFEM3D, and in Section 4 we implement the algorithm using CUDA + MPI and discuss the optimizations that we considered to try to improve efficiency. In Section 5 we validate the results of our GPU + MPI implementation with the reference C + MPI code we started from. Performance results are presented in Section 6. We conclude in Section 7.

2. Related work

For a broad range of applications, depending on the algorithms used and the ease of parallelization, applications ported to the GPU in the literature have achieved speedups that typically range from 3x to 50x with respect to calculations run on a single CPU core. Of course, whether a specific speedup is considered impressive or not depends strongly on how well the original CPU code was optimized, on the nature of the underlying computation, and also on how speedup is defined.

Dense BLAS level 3 operations, such as SGEMM and DGEMM, can be implemented to achieve close to peak arithmetic throughput, using well-known tile-based techniques [39, 40]. Large speedups are also achieved by finite-difference and finite-volume defined on structured grids (e.g., [41]). While these approaches are (mostly) limited in performance by off-chip memory bandwidth due to their low arithmetic intensity, they benefit nonetheless from up to 160 GB/s memory bandwidth on current devices, which is more than an order of magnitude higher than on high-end CPUs. Algorithms on unstructured grids are in general harder to accelerate substantially, even though Bell and Garland [42] recently demonstrated substantial speedups for various sparse matrix-vector multiply kernels and Corrigan et al. [43] presented a CFD solver on unstructured grids. The same argument holds true for higher-order methods such as spectral methods or spectral finite-element methods, which comprise many small dense operations. In both cases, performance tuning consists in balancing conflicting goals, taking into account kernel sizes, memory access patterns, sizes of small on-chip memories, multiprocessor occupancy, etc.

2.1. Geophysics and seismics on GPUs

In the area of numerical modeling of seismic wave propagation, Abdelkhalek [44], Micikevicius [41] and Abdelkhalek et al. [45] have recently used GPU computing successfully to calculate seismic reverse time migration for the oil and gas industry. They implemented a finite-difference method in the case of an acoustic medium with either constant or variable density running on a cluster of GPUs with MPI message passing. The simulation of forward seismic wave propagation or reverse time migration led to speedups of 30x and 11x, respectively. Michéa and Komatitsch [46] have also used a finite-difference algorithm on a single GPU to model forward seismic wave propagation in the case of a more complex elastic medium and obtained a speedup between 20x and 60x.

Klöckner et al. [47] implemented a discontinuous Galerkin technique to solve Maxwell's equations on a single GPU and propagate electromagnetic waves. They obtained a speedup of 50x. As mentioned in the Introduction, discontinuous Galerkin methods have also been applied to the modeling of seismic waves on CPUs [31, 32, 33, 34, 35, 36, 37]; therefore it is likely that based on the ideas of Klöckner et al. [47] to use such techniques on GPUs for Maxwell's equations they could be adapted to the modeling of seismic waves on GPUs.

Fast-multipole boundary-element methods have been applied to the propagation of seismic waves on a CPU (e.g., [48]). On the other hand, Gumerov and Duraiswami [49] have ported a general fast-multipole method to a single GPU with a speedup of 30x to 60x. Therefore, it is likely the fast-multipole boundary-element method of Chaillat et al. [48] could be implemented efficiently on a GPU.

Raghuvanshi et al. [50] studied sound propagation in a church using a Discrete Cosine Transform on a GPU and an analytical solution.

2.2. Finite elements on GPUs

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 [51, 52, 7]. The acceleration derives from a GPU implementation of a conjugate gradient

solver. A time-domain finite-element has been accelerated using OpenGL on a graphics card in [53]. The authors achieved a speedup of only two since the code is dominated by dense vector/matrix operations, an inefficient operation in OpenGL due to the lack of shared memory.

To our knowledge, the first nonlinear finite-element code ported to the GPU is in the area of surgical simulation [54]. 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 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.

2.3. GPU clusters

Currently, only a few articles have been published on the coding of applications across multiple GPUs using MPI for the same application because the initial focus was on porting application to a single GPU and maximizing performance levels. A second reason for the lack of such publications is the bottleneck that often results from communication of data between GPUs, which must pass through the PCIe bus, the CPU and the interconnect. Thus, the measured speedup of an efficient MPI-based parallel code could decrease when transformed into a multi-GPU implementation, according to Amdahl's law. The serial component of the code increases as a result of additional data transferred between the GPU and the CPU via the PCIe bus, unless the algorithm is restructured to overlap this communication with computations on the GPU. Efficiency also decreases (for a given serial component) when the time taken by the parallel component decreases, which is the case when it is accelerated via efficient GPU implementation. Again, this effect can only be avoided by reducing the serial cost of CPU to CPU and CPU to GPU communication to close to zero via overlap of computation and communication.

Fan et al. [55] described, for the first time, how an existing cluster (and an associated MPI-based distributed memory application) could be improved significantly by adding GPUs, not for visualization, but for computation. To the best of our knowledge, they were — at that time — the only group able to target realistic problem sizes, including very preliminary work on scalability and load balancing of GPU-accelerated cluster computations. Their application was a Lattice-Boltzmann solver for fluid flow in three space dimensions, using 15 million cells. As usual for Lattice-Boltzmann algorithms, single precision sufficed.

More recently, Göddeke et al. [56] used a 160-node GPU cluster and a code based on OpenGL to analyze the scalability, price/performance, power consumption, and compute density of low-order finite-element based multigrid solvers for the prototypical Poisson problem, and later extended their work to CUDA with applications from linearized elasticity and stationary laminar flow [57, 58].

GPU clusters have started to appear on the Top500 list of the 500 fastest supercomputers in the world [59]. Taking advantage of NVIDIA's Tesla architecture, several researchers have adapted their code to GPU clusters: Phillips et al. [60] have accelerated the 'NAMD' molecular dynamics program using the Charm++ parallel programming system and runtime library to communicate asynchronous one-sided messages between compute nodes. Anderson et al. [61] have reported similar success, also for molecular dynamics. Thibault and Senocak [62] used a Tesla machine (i.e., one compute node with four GPUs) to implement a finite-difference technique to solve the Navier-Stokes equations. The four GPUs were used simultaneously but message passing (MPI) was not required because they were installed on the same shared-memory compute node. Micikevicius [41] and Abdelkhalek et al. [45] used MPI calls between compute nodes equipped with GPUs to accelerate a finite-difference acoustic seismic wave propagation method. Phillips et al. [63] have accelerated an Euler solver on a GPU cluster, and recently, Stuart and Owens [64] have started to map MPI to the multiprocessors (cores) within a single GPU.

Kindratenko et al. [65] addressed various issues that they solved related to building and managing large-scale GPU clusters, such as health monitoring, data security, resource allocation and job scheduling. Fan et al. [66] and Strengert et al. [67] developed flexible frameworks to program such GPU clusters.

2.4. Precision of GPU calculations

Current GPU hardware supports quasi IEEE-754 s23e8 single precision arithmetic. Double precision is already fully IEEE-754-2008 compliant but, according to the specifications given by the vendors, theoretical peak performance is between eight (NVIDIA Tesla 10 chip), five (AMD RV870 chip) and two (NVIDIA Tesla 20 chip) times slower than single precision. However, unless the GPU code is dominated by calculation, the penalty due to double precision is far less in practice. Of course another reason to consider single precision is reduced memory storage, by a factor of exactly two.

For some finite-element applications that involve the solution of large linear systems and thus suffer from ill-conditioned system matrices, single precision is insufficient and mixed or emulated precision schemes are

desirable to achieve accurate results [68]. But our spectral-element code is sufficiently accurate in single precision, as demonstrated e.g. in Section 5 and in [69, 70, 71] and the benchmarks therein.

3. Serial algorithm

We resort to the SEM to simulate numerically the propagation of seismic waves resulting from earthquakes in the Earth or from active seismic acquisition experiments in the oil industry [69]. Another example is to simulate ultrasonic laboratory experiments [72]. The SEM solves the variational form of the elastic wave equation in the time domain on a non-structured mesh of elements, called spectral elements, in order to compute the displacement vector of any point of the medium under study.

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

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

where $\mathbf{u}$ is 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, ρ the density, and $\mathbf{f}$ an external force representing the seismic source. A colon denotes the double tensor contraction operator, a superscript T denotes the transpose, and a dot over a symbol indicates time differentiation. The material parameters of the solid, $\mathbf{C}$ and ρ, can be spatially heterogeneous and are given quantities that define the geological medium. Let us denote the physical domain of the model and its boundary by Ω and Γ respectively. We can 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,

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

The last term, i.e., the contour integral, vanishes because of the free surface boundary condition, i.e., the fact that the traction vector $\boldsymbol{\tau}=\boldsymbol{\sigma}\cdot\hat{\mathbf{n}}$ must be zero at the surface.

In a SEM, the physical domain is subdivided into mesh cells, within which quantities of interest are approximated by high order interpolants. Therefore, as in any finite-element method, a first crucial step is to design a mesh by subdividing the model volume Ω into a number of non-overlapping deformed hexahedral mesh elements Ω_e, $e=1,\ldots,n_e$, such that $\Omega=\cup_{e=1}^{n_e}\Omega_e$. For better accuracy, the edges of the elements honor the topography of the model and its main internal discontinuities, i.e., the geological layers and faults.

The mapping between Cartesian points $\mathbf{x}=(x,y,z)$ within a deformed, hexahedral element Ω_e and the reference cube may be written in the form

$$\mathbf{x}(\boldsymbol{\xi})=\sum_{a=1}^{n_a} N_a(\boldsymbol{\xi})\mathbf{x}_a. \tag{3}$$

Points within the reference cube are denoted by the vector $\boldsymbol{\xi}=(\xi,\eta,\zeta)$, where $-1\le\xi\le 1$, $-1\le\eta\le 1$ and $-1\le\zeta\le 1$. The geometry of a given element is defined in terms of $n_a=27$ control points $\mathbf{x}_a$ located in its corners as well as the middle of its edges, its faces, and in its center. The n_a shape functions N_a are triple products of degree-2 Lagrange polynomials. The three Lagrange polynomials of degree 2 with three control points $\xi_0=-1$, $\xi_1=0$, and $\xi_2=1$ are $\ell_0^2(\xi)=\frac{1}{2}\xi(\xi-1)$, $\ell_1^2(\xi)=1-\xi^2$ and $\ell_2^2(\xi)=\frac{1}{2}\xi(\xi+1)$.

A small volume $\mathrm{d}x\ \mathrm{d}y\ \mathrm{d}z$ within a given element is related to a volume $\mathrm{d}\xi\ \mathrm{d}\eta\ \mathrm{d}\zeta$ in the reference cube by $\mathrm{d}x\ \mathrm{d}y\ \mathrm{d}z=J\ \mathrm{d}\xi\ \mathrm{d}\eta\ \mathrm{d}\zeta$, where the Jacobian J of the mapping is given by $J=|\partial(x,y,z)/\partial(\xi,\eta,\zeta)|$. The partial derivative matrix $\partial\mathbf{x}/\partial\boldsymbol{\xi}$ needed for the calculation of J is obtained by analytically differentiating the mapping (3). Partial derivatives of the shape functions N_a are defined in terms of Lagrange polynomials of degree 2 and their derivatives.

To represent the displacement field in an element, the SEM uses Lagrange polynomials of degree 4 to 10, typically, for the interpolation of functions [73, 22]. Komatitsch and Tromp [70] and De Basabe and Sen [22] find that choosing the degree $n=4$ gives a good compromise between accuracy and time step duration. The $n+1$ Lagrange polynomials of degree n are defined in terms of $n+1$ control points $-1\le\xi_\alpha\le 1$, $\alpha=0,\ldots,n$, by

$$\ell_\alpha^n(\xi)=\frac{(\xi-\xi_0)\cdots(\xi-\xi_{\alpha-1})(\xi-\xi_{\alpha+1})\cdots(\xi-\xi_n)}{(\xi_\alpha-\xi_0)\cdots(\xi_\alpha-\xi_{\alpha-1})(\xi_\alpha-\xi_{\alpha+1})\cdots(\xi_\alpha-\xi_n)}\,. \tag{4}$$

The control points ξ_α are chosen to be the $n+1$ Gauss-Lobatto-Legendre (GLL) points, which are the roots of $(1-\xi^2)P'_n(\xi) = 0$, where P'_n denotes the derivative of the Legendre polynomial of degree n [74, p. 61]. The reason for this choice is that the combination of Lagrange interpolants with GLL quadrature greatly simplifies the algorithm because the mass matrix becomes diagonal and therefore permits the use of fully explicit time schemes [69], which can be implemented efficiently on large parallel machines [e.g., 25].

Functions f that represent the physical unknowns on an element are interpolated in terms of triple products of Lagrange polynomials of degree n as

$$f(\mathbf{x}(\xi,\eta,\zeta)) \approx \sum_{\alpha,\beta,\gamma=0}^{n_\alpha,n_\beta,n_\gamma} f^{\alpha\beta\gamma}\ell_\alpha(\xi)\ell_\beta(\eta)\ell_\gamma(\zeta), \tag{5}$$

where $f^{\alpha\beta\gamma} = f(\mathbf{x}(\xi_\alpha,\eta_\beta,\zeta_\gamma))$ denotes the value of the function f at the GLL point $\mathbf{x}(\xi_\alpha,\eta_\beta,\zeta_\gamma)$. The gradient of a function, $\boldsymbol{\nabla} f = \sum_{i=1}^{3} \hat{\mathbf{x}}_i \partial_i f$, evaluated at the GLL point $\mathbf{x}(\xi_{\alpha'},\eta_{\beta'},\zeta_{\gamma'})$, can thus be written as

$$\begin{aligned}
\boldsymbol{\nabla} f(\mathbf{x}(\xi_{\alpha'},\eta_{\beta'},\zeta_{\gamma'})) \;\approx\; & \sum_{i=1}^{3} \hat{\mathbf{x}}_i \Bigg[(\partial_i\xi)^{\alpha'\beta'\gamma'} \sum_{\alpha=0}^{n_\alpha} f^{\alpha\beta'\gamma'}\ell'_\alpha(\xi_{\alpha'}) \\
& + (\partial_i\eta)^{\alpha'\beta'\gamma'} \sum_{\beta=0}^{n_\beta} f^{\alpha'\beta\gamma'}\ell'_\beta(\eta_{\beta'}) \\
& + (\partial_i\zeta)^{\alpha'\beta'\gamma'} \sum_{\gamma=0}^{n_\gamma} f^{\alpha'\beta'\gamma}\ell'_\gamma(\zeta_{\gamma'}) \Bigg].
\end{aligned} \tag{6}$$

Here, $\hat{\mathbf{x}}_i$, $i = 1,2,3$, denote unit vectors in the directions of increasing x, y, and z, respectively, and ∂_i, $i = 1,2,3$, denote partial derivatives in those directions. We use a prime to denote derivatives of the Lagrange polynomials, as in ℓ'_α.

To solve the weak form of the equations of motion (2) requires numerical integrations over the elements. A GLL integration rule is used:

$$\begin{aligned}
\int_{\Omega_e} f(\mathbf{x})\ \mathrm{d}^3\mathbf{x} \;&=\; \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} f(\mathbf{x}(\xi,\eta,\zeta))\, J(\xi,\eta,\zeta)\ \mathrm{d}\xi\ \mathrm{d}\eta\ \mathrm{d}\zeta \\
&\approx\; \sum_{\alpha,\beta,\gamma=0}^{n_\alpha,n_\beta,n_\gamma} \omega_\alpha\omega_\beta\omega_\gamma f^{\alpha\beta\gamma} J^{\alpha\beta\gamma}\ ,
\end{aligned} \tag{7}$$

where $J^{\alpha\beta\gamma} = J(\xi_\alpha,\eta_\beta,\zeta_\gamma)$, and $\omega_\alpha > 0$, for $\alpha = 0,\ldots,n$, denote the weights associated with the GLL quadrature [74, p. 61]. Therefore in our case each spectral element contains $(n+1)^3 = 125$ GLL points.

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

$$\mathrm{M}\ddot{\mathrm{U}} + \mathrm{K}\mathrm{U} = \mathrm{F}\ , \tag{8}$$

where U is the displacement vector we want to compute, 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 for instance [69]. Time integration of this system is usually performed based on a second-order centered finite-difference Newmark time scheme (e.g., [75, 70]), although higher-order time schemes can be used if necessary [76, 77].

In the SEM algorithm, the serial time loop dominates the total cost because in almost all wave propagation applications a large number of time steps is performed, typically between 5,000 and 100,000. The preprocessing and postprocessing phases are negligible in terms of cost and it is therefore sufficient to focus on the time loop to optimize a SEM code. In addition, all the time steps have identical cost because the mesh is static and the algorithm is fully explicit, which greatly facilitates optimization.

4. Implementation on a cluster of GPUs using CUDA and non-blocking MPI

4.1. CUDA programming model

For readers not familiar with details of CUDA, we briefly explain the programming model that supports the fine-grained parallel architecture of NVIDIA GPUs. We refer to the CUDA documentation [8] and conference tutorials (http://gpgpu.org/developer) for details.

CUDA-related publications (see Section 2), the official CUDA documentation, and various press releases vary dramatically in terminology, especially when referring to the notion of a 'core' on GPUs. In this article, we follow the classification of Fatahalian and Houston [12] and identify each multiprocessor with a 'SIMD core'. The individual thread processors ('streaming processor cores' or more recently 'CUDA cores' in NVIDIA nomenclature) within each multiprocessor share the instruction stream, and it is therefore justified to view them as arithmetic logic units (ALUs) or SIMD units.

CUDA 'kernels' are executed by partitioning the computation into a 'grid' of 'thread blocks'. The atomic execution unit is thus the thread block, which corresponds to a virtualized multiprocessor. The same physical multiprocessor can execute several blocks, and the order in which blocks are assigned to multiprocessors is undefined. Threads within one block exchange data via a small low-latency on-chip 'shared memory'. Synchronization between blocks is only possible at the kernel scope, i.e., there is always an implicit synchronization between kernel calls on dependent data (i.e., when some of the output of one kernel is used as input to the next). It is not possible (except via slow atomic memory operations which are currently not available for floating point data) to synchronize blocks within the execution of one grid.

All threads within a single block are executed in a SIMD fashion (NVIDIA refers to this as SIMT, single instruction multiple threads), and the SIMD granularity is the 'warp'. The threads within one block are assigned to consecutive warps of 32 threads. The ALUs within a multiprocessor execute an entire warp in lockstep, using a shared instruction pointer. Branch divergence within a warp should therefore be avoided to prevent a potentially large performance hit since all threads within one warp execute both sides of the branch and unused results are masked out.

Accesses to global memory can be 'coalesced' automatically by the hardware into, at best, a single large, efficient transaction per 'half-warp' of 16 threads. To achieve that, there are a number of restrictions on the memory access pattern and on data alignment. On the Tesla GPUs we employ in this article, global memory can be conceptually organized into a sequence of memory-aligned 128-byte segments (for single precision data, which we use). The number of memory transactions performed for a half-warp is equal to the number of 128-byte segments touched by the addresses used by that half-warp. Only 64 bytes are retrieved from memory if all the addresses of the half-warp lie in the upper or lower half of a memory segment, with a doubling of the bandwidth. Fully coalesced memory requests and thus maximum memory performance are achieved if all addresses within a half-warp touch precisely the upper or lower half of a single segment (see Bell and Garland [42] for an example based on strided memory accesses). The precise conditions for maximum throughput are a function of the graphics card, the precision, and the compute capability of the card. Each multiprocessor can keep 1024 threads in flight simultaneously, by switching to the next available warp when the currently executed one stalls, e.g., due to memory requests. These context switches are performed in hardware and are thus essentially for free. The actual number of concurrent thread blocks is determined by the amount of resources used, in particular registers and shared memory (referred to as 'multiprocessor occupancy'). This approach maximizes the utilization of the ALUs, minimizes the effective latency of off-chip memory requests, and thus maximizes the overall throughput. Table 1 summarizes some of the above for easier reference.

Term	**Explanation**
Host	The CPU and the interconnect.
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– all the threads within a block can be synchronized.
Grid	A set of thread blocks - a kernel is executed on a grid of thread blocks.
Warp	A group of 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, essentially a measure of latency hiding.
Global memory	Uncached off-chip DRAM memory.
Shared memory	High-performance on-chip register memory, limited to 16 kB per multiprocessor on the hardware we use.
Constant memory	A read-only region of device memory with faster access times and a cache mechanism.
Coalesced memory accesses	Simultaneous GPU global memory accesses coalesced into a single contiguous, aligned memory access at the scope of a half-warp.

Table 1: Glossary of some terms used in CUDA programming and in this article. For more details the reader is referred to the CUDA documentation [8] and conference tutorials (http://gpgpu.org/developer).

4.2. Meshing and partitioning into subdomains

In a preprocessing step on the CPU, we mesh the region of the Earth in which the earthquake occurred (Figure 1, left). Next, we split the mesh into slices, one per computing unit/MPI process (one per processor core on a CPU cluster and one per GPU on a GPU cluster) (Figure 1, right). The mesh in each slice is unstructured in the finite-element sense, i.e., it has a non regular topology and the valence of a point of the mesh can be greater than eight, which is the maximum value in a regular mesh of hexahedra. Equivalently we can say that mesh elements can have any number of neighbors, and that this number varies across a mesh slice.

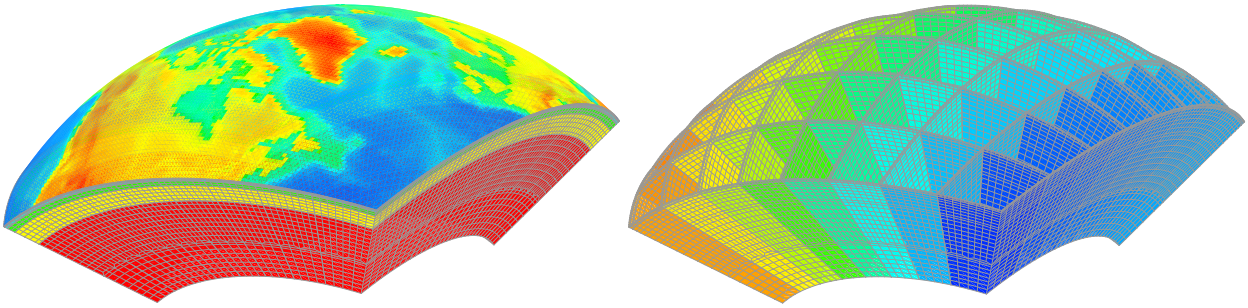

Figure 1: (Left) Mesh of a region of the Earth centered on the North pole, showing Greenland and part of North America. The mesh is unstructured, as can also been seen on the closeups of Figures 3 and 6. (Right) The mesh is decomposed into 64 slices in order to run on 64 GPUs. The decomposition is topologically regular, i.e., all the mesh slices and the cut planes have the same number of elements and points, and the cut planes all have the same structure. In reality the mesh is filled with 3D elements, but only the cut planes have been drawn for clarity.

However, the whole mesh is block-structured, i.e., each mesh slice is composed of an unstructured mesh, but all the mesh slices are topologically identical. In other words, the whole mesh is composed of a regular pattern of identical unstructured slices. Therefore when we split the mesh into slices the decomposition is topologically a regular grid and all the mesh slices and the cut planes have the same number of elements and points. This implies that perfect load balancing is ensured by definition between all the MPI tasks.

4.3. Assembling common points

The Lagrange interpolants, defined on $[-1, 1]$, are built from the Gauss-Lobatto-Legendre points, which include the boundary points -1 and +1 in each coordinate direction. Polynomial basis functions of degree n also include $(n-1)$ interior points. In 3D this holds true in the three spatial directions. Therefore, $(n-1)^3$ points are interior points not shared with neighboring elements in the mesh, and $(n+1)^3 - (n-1)^3$ may be shared with neighboring elements through a common face, edge or corner, as illustrated in Figure 2. One can thus view the mesh either as a set of elements, each with $(n+1)^3$ points, or as a set of global grid points with all the multiples, i.e., the common points, counted only once. Thus, some bookkeeping is needed to store an array that describes the mapping between a locally numbered system for all the elements and their associated grid points in a global numbering system.

When computing elastic mechanical forces with a SEM, the contributions to the force vector are calculated locally and independently inside each element and summed at the shared points. This is called the 'assembly process' in finite-element algorithms. In a parallel code, whether on CPUs or on GPUs, this operation is the most critical one and has the largest impact on performance. We use below an efficient approach to the assembly process based on mesh coloring. Once assembled, the global force vector is scaled by the inverse of the assembled mass matrix, which is diagonal and therefore stored as a vector (see Section 3). This last step is straightforward and has negligible impact on performance.

4.4. Overlapping computation and communication

The elements that compose the mesh slices of Figure 1 (right) are in contact through a common face, edge or point. To allow for overlap of communication between cluster nodes with calculations on the GPUs, we create inside each slice a list of all the elements that are in contact with any other mesh slice through a common face, edge or point. Members of this list are called 'outer' elements; all other elements are called 'inner' elements (Figure 3). We compute the outer elements first, as it is done classically (cf. for instance Danielson and Namburu [78], Martin et al. [26], Micikevicius [41], Michéa and Komatitsch [46]). Once the computation of the outer elements is complete, we can fill the MPI buffers and issue a non-blocking MPI call, which initiates the communication and returns immediately. While the MPI messages are traveling

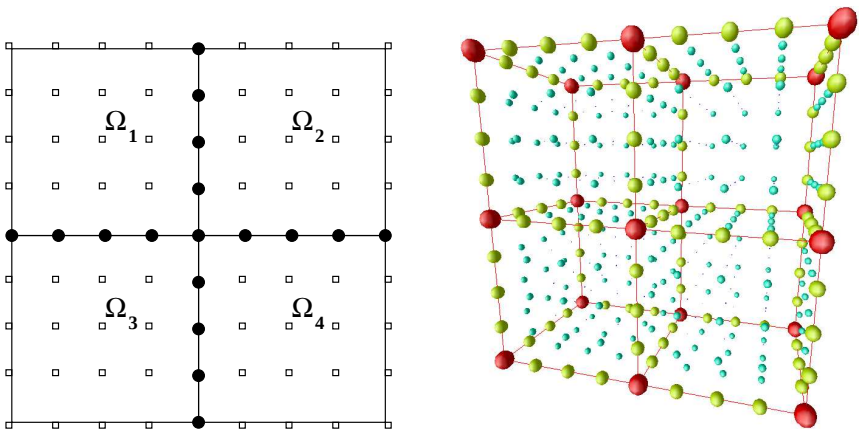

Figure 2: (Left) In a SEM mesh in 2D, elements can share an edge or a corner. (Right) In 3D, elements can share points on a face, an edge or a corner. The GLL interpolation and quadrature points inside each element are non-evenly spaced but have been drawn evenly-spaced for clarity.

across the network, we compute the inner elements. Achieving effective overlap requires that the ratio of the number of inner to outer elements be sufficiently large, which is the case for large enough mesh slices. Under these conditions, the MPI data transfer will statistically likely complete before the completion of the computation of the inner elements. We note that to achieve effective overlap on a cluster of GPUs, this ratio must be larger than what is required for a classical cluster of CPUs, since calculation of the inner elements is over an order of magnitude faster when executed on a GPU (see Section 6).

4.5. System design

Data transfers between CPU and GPU decrease the efficiency of an implementation because of the limited bandwidth of the PCIe bus. To mitigate this potential bottleneck, we allocate and load all the local and global arrays on the GPU prior to the start of the time loop, which is appropriate because the operations performed are identical at each iteration. As a consequence, the total problem size is limited by the amount of memory available on each GPU, i.e., 4 GB on the Tesla S1070 models used in this study. For mesh slices too large to fit on 4 GB per GPU, and under the assumption that that there is more than 4 GB of memory per MPI task on the CPU nodes, another alternative is an implementation that follows the second algorithm presented in [71], in which local arrays are stored on the CPU and sent in batches to the GPU to be processed. The disadvantage of this approach is the large number of data transferred back and forth between CPU and GPU, with a large impact on performance and increased algorithmic complexity.

The high 25x speedup achieved on a single GPU versus a calculation running on a single processor core [71] diminishes some of the advantages of designing a mixed CPU/GPU system in which some fraction of the calculations would execute on the CPU. Aside from a factor 25 slowdown, additional costs would arise from significant data transfers back and forth between the CPU and the GPU, for instance to assemble the mechanical forces. Let us estimate, under the best possible conditions, the expected additional speedup possible by harnessing the power of the multiple cores to help update additional elements. Thus, we assume that 1) there are no communication costs between GPU and CPU or between cores, and 2) there is perfect load balancing, i.e., cores and GPUs never wait on each other. We further assume that the acceleration of a single GPU over a single CPU, to compute a single element C is A, and that the wall clock time on a dedicated machine to compute a single element is C, and that there are E elements on the GPU. With the assumption of load balancing, each CPU core holds E/A elements, and the time to compute the problem on n CPU cores and m GPUs is CE. On the other hand, the time to update all the elements on a single GPU is: $C(m + n/A)E$. Thus, the best case acceleration is $m + n/A$. Taking the parameters that correspond to our architectural configuration, there is 1 GPU for every two CPU cores. Therefore, taking $A = 25$, the possible acceleration beyond that provided by a single GPU is 8%. Achieving a factor of two would require 25 cores. Note that in reality, communication costs, PCIe bus sharing, and imperfect load-balancing will decrease the computed gain in performance. The additional complexities led us to purposely use only one CPU core per GPU (instead of a maximum of 2) and perform on it only the parts of the algorithm that must remain on the CPU, i.e., the handling and processing of the MPI buffers.

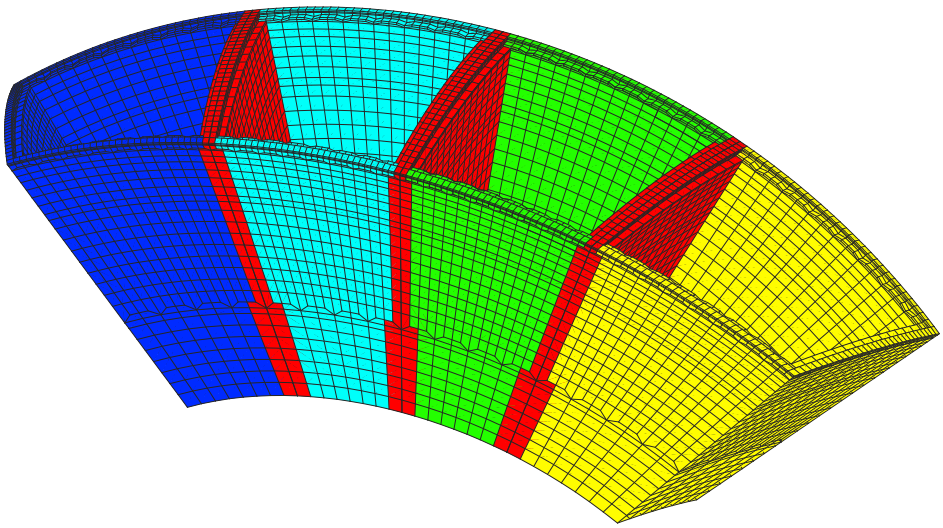

Figure 3: Outer (red) and inner (other colors) elements for part of the mesh of Figure 1. Elements in red have at least one point in common with an element from another slice and must therefore be computed first, before initiating the non-blocking MPI communications. The picture also shows that the mesh is unstructured because we purposely increase the size of the elements with depth based on a 'mesh doubling brick'. This is done because seismic velocities and therefore seismic wavelengths increase with depth in the Earth. Therefore, larger elements are sufficient to sample them; they lead to a reduction in the computational cost. Here for clarity only the cut planes that define the mesh slices have been drawn; but in reality the mesh is filled (see Figure 1, left, and Figure 6).

4.6. CUDA implementation

The computations performed during each iteration of the SEM consist of three steps. The first step performs a partial update of the global displacement vector $\mathbf{u}$ and velocity vector $\mathbf{v} = \dot{\mathbf{u}}$ in the Newmark time scheme at all the grid points, based on the previously computed acceleration vector $\mathbf{a} = \ddot{\mathbf{u}}$:

$$\mathbf{u}^{\text{new}} = \mathbf{u}^{\text{old}} + \Delta t\, \mathbf{v} + \frac{\Delta t}{2}\, \mathbf{a} \tag{9}$$

and

$$\mathbf{v}^{\text{new}} = \mathbf{v}^{\text{old}} + \frac{\Delta t}{2}\, \mathbf{a}\ , \tag{10}$$

where Δt is the time step.

The second step is by far the most complex. It consists mostly of local matrix products inside each element to compute its contribution to the stiffness matrix term KU of eq. (8) according to eq. (2). To do so, the global displacement vector is first copied into each element using the local-to-global mesh numbering mapping defined above. Then, matrix products must be performed between a derivative matrix, whose components are the derivatives of the Lagrange polynomials at the GLL points $\ell'_\alpha(\xi_{\alpha'})$ as in eq. (6), and the displacement $\mathbf{u}$ in 2D cut planes along the three local directions (i, j, k) of an element.

The goal of this step is to compute the gradient of the displacement vector at the local level. Subsequently the numerical integrations of eq. (7) are performed. They involve the GLL integration weights ω_α and the discrete Jacobian at the GLL points, $J(\xi_\alpha, \eta_\beta, \zeta_\gamma)$. To summarize, we compute

$$\begin{aligned}\int_{\Omega_e} \boldsymbol{\nabla}\mathbf{w} : \boldsymbol{\sigma}\ \mathrm{d}^3\mathbf{x} \quad \approx \quad & \sum_{\alpha,\beta,\gamma=0}^{n_\alpha,n_\beta,n_\gamma} \sum_{i=1}^{3} w_i^{\alpha\beta\gamma} \Bigg[\omega_\beta \omega_\gamma \sum_{\alpha'=0}^{n_{\alpha'}} \omega_{\alpha'} J_e^{\alpha'\beta\gamma} F_{i1}^{\alpha'\beta\gamma} \ell'_\alpha(\xi_{\alpha'}) \\ + \quad & \omega_\alpha \omega_\gamma \sum_{\beta'=0}^{n_{\beta'}} \omega_{\beta'} J_e^{\alpha\beta'\gamma} F_{i2}^{\alpha\beta'\gamma} \ell'_\beta(\eta_{\beta'}) + \omega_\alpha \omega_\beta \sum_{\gamma'=0}^{n_{\gamma'}} \omega_{\gamma'} J_e^{\alpha\beta\gamma'} F_{i3}^{\alpha\beta\gamma'} \ell'_\gamma(\zeta_{\gamma'}) \Bigg]\ ,\end{aligned} \tag{11}$$

where

$$F_{ik} = \sum_{j=1}^{3} \sigma_{ij}\, \partial_j \xi_k, \tag{12}$$

and $F_{ik}^{\sigma\tau\nu} = F_{ik}(\mathbf{x}(\xi_\sigma, \eta_\tau, \zeta_\nu))$ denotes the value of F_{ik} at the GLL point $\mathbf{x}(\xi_\sigma, \eta_\tau, \zeta_\nu)$. For brevity, we have introduced index notation ξ_i, $i = 1, 2, 3$, where $\xi_1 = \xi$, $\xi_2 = \eta$, and $\xi_3 = \zeta$. In index notation, the elements

of the Jacobian matrix $\partial\boldsymbol{\xi}/\partial\mathbf{x}$ may be written as $\partial_i\xi_j$. The value of the stress tensor $\boldsymbol{\sigma}$ at the GLL points is determined by

$$\boldsymbol{\sigma}(\mathbf{x}(\xi_\alpha,\eta_\beta,\zeta_\gamma),t) = \mathbf{C}(\mathbf{x}(\xi_\alpha,\eta_\beta,\zeta_\gamma)) : \boldsymbol{\nabla}\mathbf{u}(\mathbf{x}(\xi_\alpha,\eta_\beta,\zeta_\gamma),t) \ . \tag{13}$$

This calculation requires knowledge of the gradient $\boldsymbol{\nabla}\mathbf{u}$ of the displacement at these points, which, using (eq. 6), is

$$\begin{aligned}\partial_i u_j(\mathbf{x}(\xi_\alpha,\eta_\beta,\zeta_\gamma),t) &= \left[\sum_{\sigma=0}^{n_\sigma} u_j^{\sigma\beta\gamma}(t)\ell'_\sigma(\xi_\alpha)\right]\partial_i\xi(\xi_\alpha,\eta_\beta,\zeta_\gamma) + \left[\sum_{\sigma=0}^{n_\sigma} u_j^{\alpha\sigma\gamma}(t)\ell'_\sigma(\eta_\beta)\right]\partial_i\eta(\xi_\alpha,\eta_\beta,\zeta_\gamma) \\ &+ \left[\sum_{\sigma=0}^{n_\sigma} u_j^{\alpha\beta\sigma}(t)\ell'_\sigma(\zeta_\gamma)\right]\partial_i\zeta(\xi_\alpha,\eta_\beta,\zeta_\gamma) \ . \end{aligned} \tag{14}$$

Finally at the end of this second main calculation the computed local values are summed ('assembled') at global mesh points to compute the acceleration vector $\mathbf{a}$ using the local-to-global mesh numbering mapping.

The third main calculation performs the same partial update of the global velocity vector in the Newmark time scheme at all the grid points as in e.q (10), based on the previously computed acceleration vector. It cannot be merged with it because the acceleration vector is modified in the second main calculation.

Figure 4 summarizes the structure of a single time step; this structure contains no significant serial components. Therefore Amdahl's law, which says that any large serial part will drastically reduce the potential speedup of any parallel application, does not have a significant impact in our SEM application.

Implementation of the first and third kernels. Each of the three main calculations is implemented as an individual CUDA kernel. The first and third kernels are not detailed here because they are straightforward: they consist of a simple local calculation at all the global grid points, without dependencies. These calculations are trivially parallel and reach 100% occupancy in CUDA. As one element contains $(n+1)^3 = 125$ grid points, we use zero padding to create thread blocks of 128 points, i.e., we use one block per spectral element because in CUDA blocks should contain an integral multiple of 32-thread warps for maximum performance. This improves performance by achieving automatic memory alignment on multiples of 16. The threads of a half-warp load adjacent elements of a float array and access to global memory is thus perfectly coalesced, i.e., optimal. This advantage outweighs by far wasting $128/125 = 2.4\%$ of the memory.

Implementation of the second kernel. The second kernel is illustrated in Figure 5 and is much more complex. A key issue is how to properly handle the summation of material elastic forces computed in each element. As already noted, some of these values get summed at shared grid points and therefore in principle the sum should be atomic. The idea is to ensure that different warps never update the same shared location, which would lead to incorrect results. Current NVIDIA hardware and CUDA support atomic operations, but only for integers, not floating point data. In addition, in general atomic memory operations can be inefficient because memory accesses are serialized. We therefore prefer to create subsets of disjoint mesh elements based on mesh coloring [79, 80, 81, 82, 71] as shown in Figure 6. In a coloring strategy, elements of different colors have no common grid points, and the update of elements in a given set can thus proceed without atomic locking. Adding an outer serial loop over the mesh colors, each color is handled through a call to the second CUDA kernel, as shown in the flowchart of a single time step in Figure 4. Mesh coloring is done once in a preprocessing stage when we create the mesh on the CPU nodes (see section 4.2). It is therefore not necessary to port this preprocessing step to CUDA. For the same reason, its impact on the numerical cost of a sufficiently long SEM simulation is small.

We use the same zero padding strategy and mapping of each element to one thread block of 128 threads with one thread per grid point as for kernels one and three. The derivative matrices used in kernel 2 have size $(n+1)\times(n+1)$, i.e., 5×5. We inline these small matrix products manually, and store them in constant memory to take advantage of its faster access times and cache mechanism. All threads of a half-warp can access the same constant in one cycle.

Kernel 2 dominates the computational cost and is memory bound because of the small size of the matrices involved: it performs a relatively large number of memory accesses compared to a relatively small number of calculations. Furthermore, we have to employ indirect local-to-global addressing, which is intrinsically related to the unstructured nature of the mesh and leads to some uncoalesced memory access patterns. On the GT200 architecture, we nonetheless see very good throughput of this kernel. On older hardware (G8x and G9x chips) the impact on performance was much more severe [71].

We also group the copying of data in and out of the MPI buffers to minimize the bottleneck imposed by the PCIe bus: we collect all the data to be transferred to the CPU into a single buffer to execute one large PCIe transfer because a small number of large transfers is better than many smaller transfers.

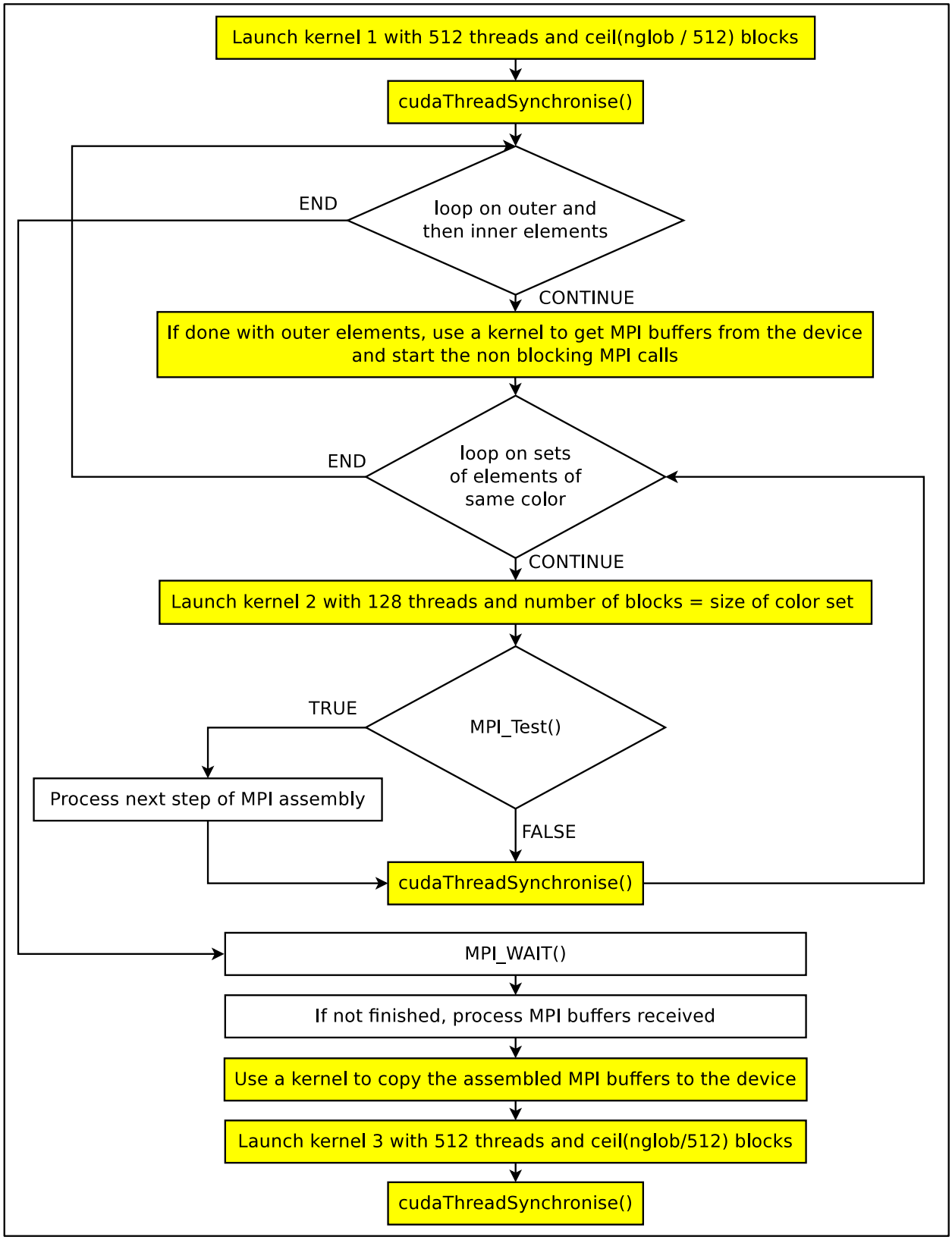

Figure 4: Implementation of one time step of the main serial time loop of our spectral-element method implemented in CUDA + MPI on a GPU cluster. The white boxes are executed on the CPU, while the filled yellow boxes are launched as CUDA kernels on the GPU. Some boxes correspond to transfers between the CPU and the GPU, or between the GPU and the CPU, in order to copy and process the MPI buffers.

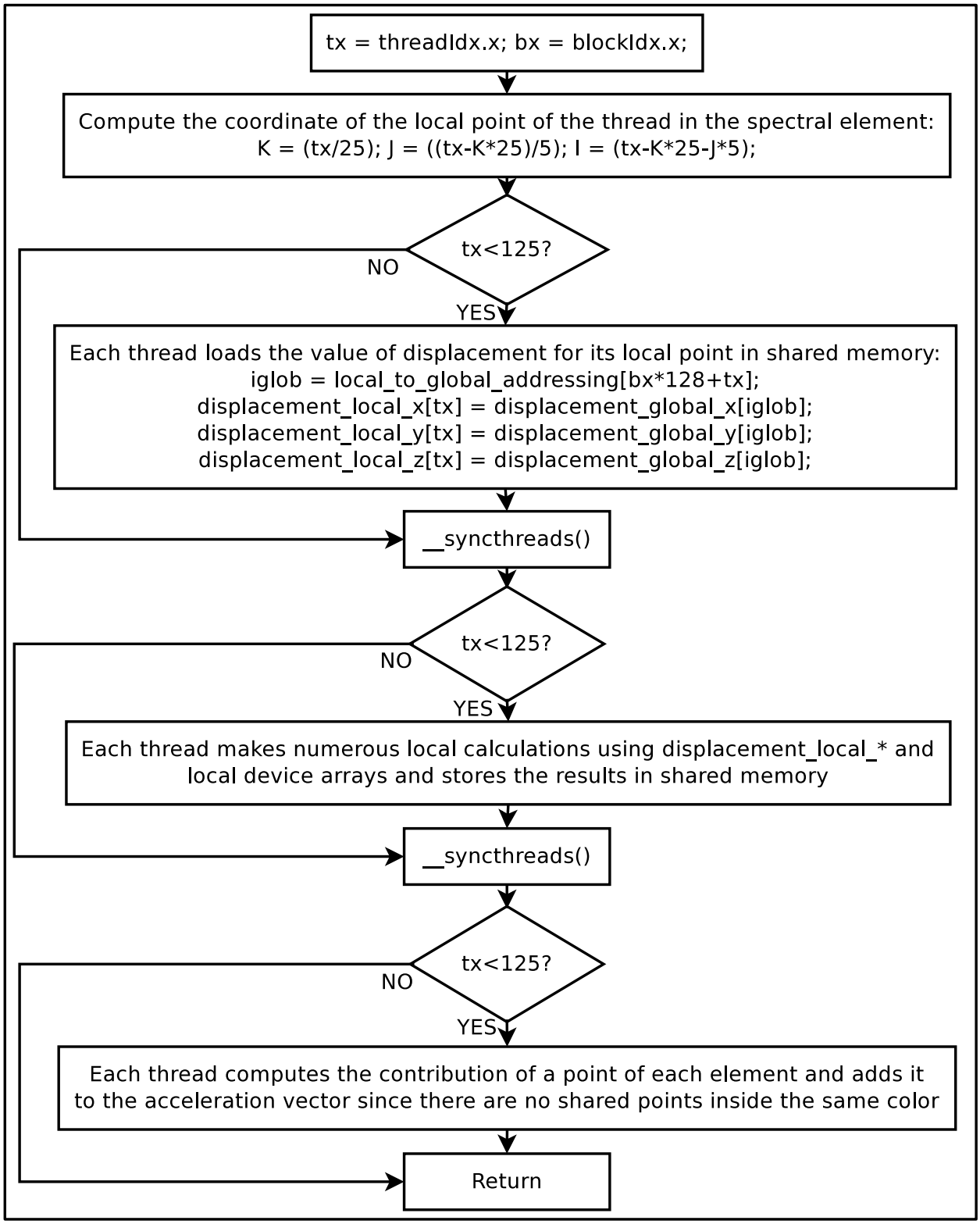

Figure 5: Flowchart of kernel 2, which is the most complex kernel. It implements the global-to-local copy of the displacement vector, then matrix multiplications between the local displacement and some derivative matrix, then multiplications with GLL numerical integration coefficients, and then a local-to-global summation at global mesh points, some of which being shared between adjacent mesh elements of a different color.

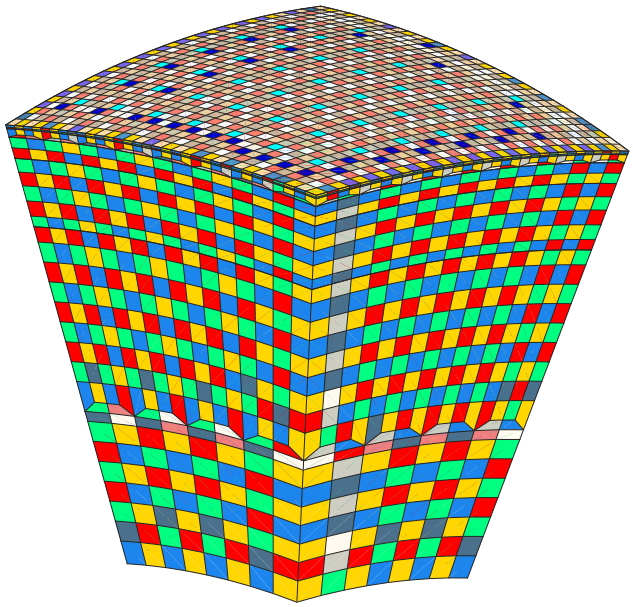

Figure 6: Mesh coloring allows us to create subsets of elements (in each color) that do not share mesh points. Therefore, each of these subsets can be handled efficiently on a GPU without resorting to an atomic operation when summing values at the mesh points. Because we process the 'outer' and the 'inner' elements of Figure 3 in two separate steps, we color them separately.

5. Numerical validation

5.1. Comparison to a reference solution in C + MPI

We first need to make sure that our CUDA + non-blocking MPI implementation works fine. Let us therefore perform a validation test in which we compare the results given by the new code with the existing single-precision C + MPI version of the code for a CPU cluster, which has been used for many years and is fully validated (e.g., [69, 70]). We take the mesh structure of Figure 1 (left) composed of 64 mesh slices and put a vertical force source at latitude 42.17°, longitude -50.78° and a depth of 1837.8 km. The model of the structure of the Earth is the elastic isotropic version of the PREM model without the ocean layer [83], which is a standard reference Earth model widely used in the geophysical community. The time variation of the source is the second derivative of a Gaussian with a dominant period of 50 s, centered on time $t = 60$ s. The mesh is composed of 256×256 spectral elements at the surface and contains a total of 655,360 spectral elements and 43,666,752 independent grid points. We use a time step of 0.20 s to honor the CFL stability condition [70, 22, 77] and propagate the waves for a total of 10,000 time step, i.e., 2,000 s. We record the time variation (due to the propagation of seismic waves across the mesh) of the three components of the displacement vector (a so-called 'seismogram') at latitude 37.42°, longitude 139.22° and a depth of 22.78 km.

Figure 7 shows that the three seismograms are indistinguishable at the scale of the figure and that the difference is very small. This difference is due to the fact that operations are performed in a different order on a GPU and on a CPU and thus cumulative roundoff is slightly different, keeping in mind that floating point arithmetic is not associative.

5.2. Application to a real earthquake in Bolivia

Let us now study a real earthquake of magnitude $M_w = 8.2$ that occurred in Bolivia on June 9, 1994, at a depth of 647 km. The real data recorded near the epicenter during the event by the so-called 'BANJO' array of seismic recording stations have been analyzed for instance by Jiao et al. [84]. The earthquake was so large that a permanent displacement (called a 'static offset') of 6 to 7 mm was observed in a region several hundred kilometers wide, i.e., the surface of the Earth was permanently tilted and the displacement seismograms do not go back to zero. This was confirmed by Ekström [85] who used a quasi-analytical calculation based on the spherical harmonics of the Earth (the so-called 'normal-mode summation technique') for a 1-D spherically-symmetric Earth model and also found this static offset. Let us therefore see if our SEM calculation in CUDA + MPI can calculate it accurately as well.

The mesh and all the numerical parameters are unchanged compared to Figure 7, except that the mesh is centered on Bolivia. The simulation is accurate for all seismic periods greater than about 15 s (i.e., all seismic frequencies below 1/15 Hz). In Figure 8 we show the SEM seismograms at seismic recording station 'ST04'

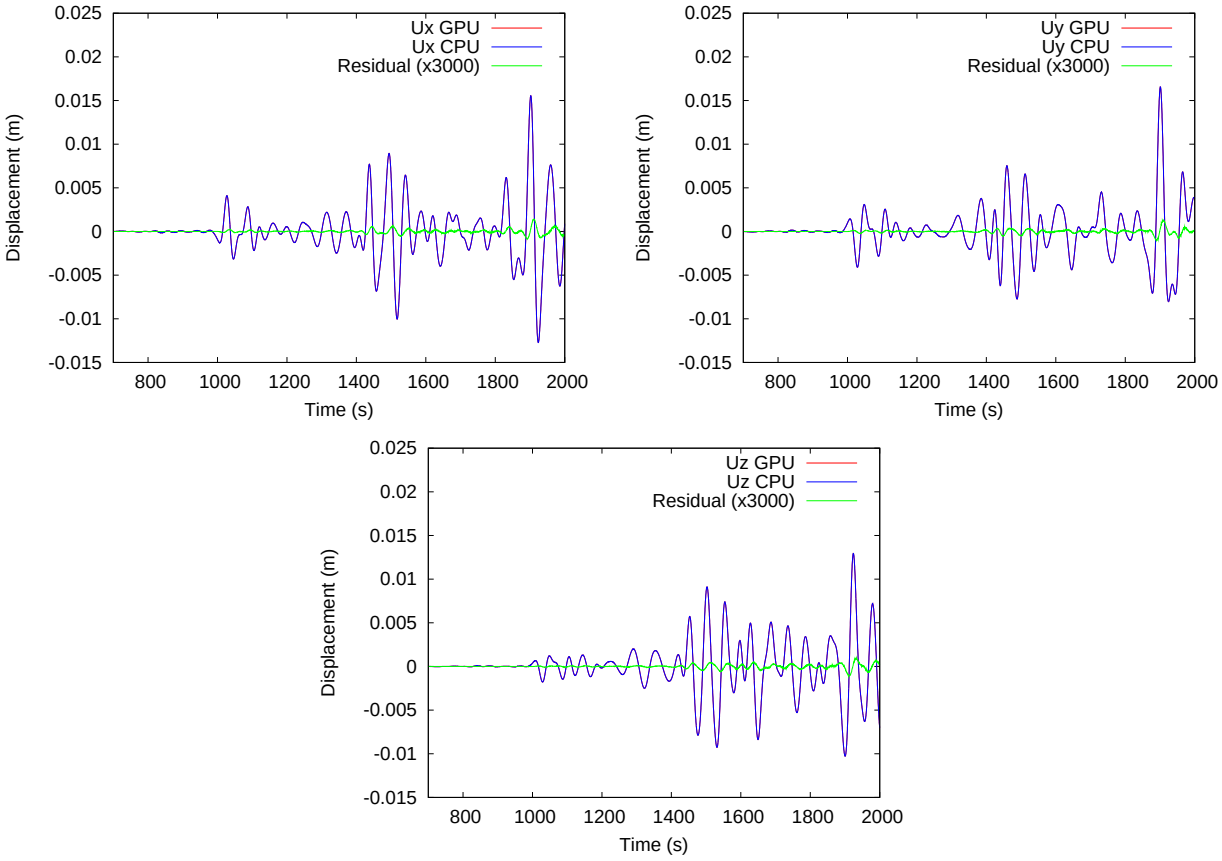

Figure 7: Comparison of the time variation (due to the propagation of seismic waves across the mesh) of the three components of the displacement vector (upper left: X component; upper right: Y component; bottom: Z component) at a given point of the mesh due to the presence of a vertical force source elsewhere in the mesh, for the single-precision CUDA + MPI code (red line) and the reference existing single-precision C + MPI code running on the CPUs (blue line). The two curves are indistinguishable at the scale of the figure. The absolute difference (shown amplified by a factor of 3000, green line) is very small, which validates our CUDA + MPI implementation.

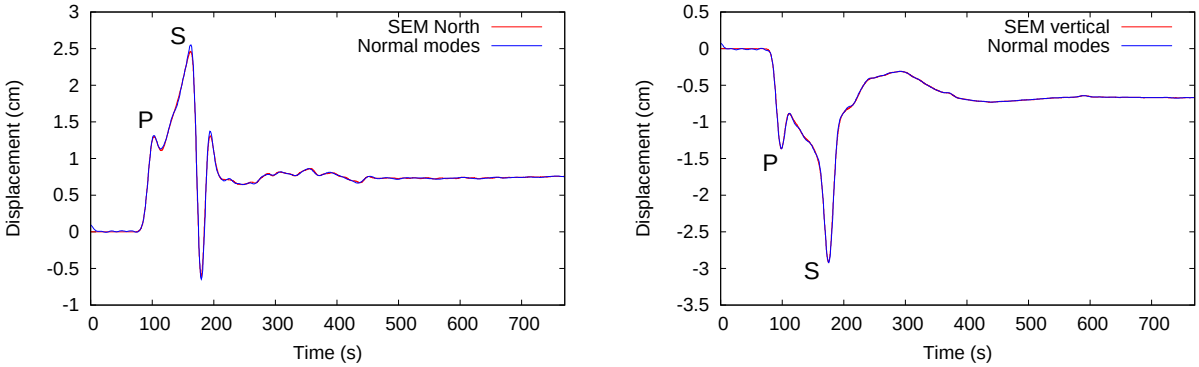

Figure 8: Numerical time variation of the components of the displacement vector at station 'ST04' of the 'BANJO' seismic recording array in Bolivia during the magnitude 8.2 earthquake of June 9, 1994. The earthquake occurred at a depth of 647 km. Left: North-South component, Right: vertical component. The red line shows the spectral-element CUDA + MPI solution and the blue line shows the independent quasi-analytical reference solution obtained based on normal-mode summation. The agreement is excellent, which further validates our CUDA + MPI implementation. The pressure (P) and shear (S) waves are accurately computed and the large 6.6 mm and 7.3 mm static offsets observed on the vertical and North-South components, respectively, are correctly reproduced. Note that the quasi-analytical reference solution obtained by normal-mode summation contains a small amount of non-causal numerical noise (see e.g. the small oscillations right after $t = 0$ on the North component, while the first wave arrives around 100 s). Therefore, the small discrepancies are probably due to the reference solution.

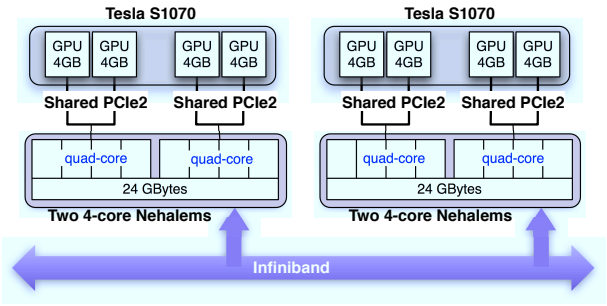

Figure 9: Description of the cluster of 48 Teslas S1070 used in this study. Each Tesla S1070 has four GT200 GPUs and two PCI Express-2 buses (i.e., two GPUs share a PCI Express-2 bus). The GT200 cards have 4 GB of memory, and the memory bandwidth is 102 gigabytes per second with a memory bus width of 512 bit. The Teslas are connected to BULL Novascale R422 E1 nodes with two quad-core Intel Xeon X5570 Nehalem processors operating at 2.93 GHz. Each node has 24 GB of RAM. The network is a non-blocking, symmetric, full duplex Voltaire InfiniBand double data rate (DDR) organized as a fat tree.

of the BANJO array in Bolivia, which is located at a distance of 5° south of the epicenter. Superimposed we show the independent quasi-analytical solution computed based on normal-mode summation. The agreement is excellent, which further validates our CUDA + MPI code. The strong pressure (P) and shear (S) wave arrivals are correctly reproduced and the 6.6 mm offset on the vertical component and 7.3 mm on the North-South component are computed accurately.

6. Performance analysis and speedup obtained

The machine we use is a cluster of 48 Teslas S1070 at CCRT/CEA/GENCI in Bruyères-le-Châtel, France; each Tesla S1070 has four GT200 GPUs and two PCI Express-2 buses (i.e., two GPUs share a PCI Express-2 bus). The GT200 cards have 4 GB of memory, and the memory bandwidth is 102 gigabytes per second with a memory bus width of 512 bits. The Teslas are connected to BULL Novascale R422 E1 nodes with two quad-core Intel Xeon X5570 Nehalem processors operating at 2.93 GHz. Each node has 24 GB of RAM and runs Linux kernel 2.6.18. The network is a non-blocking, symmetric, full duplex Voltaire InfiniBand double data rate (DDR) organized as a fat tree. Figure 9 illustrates the cluster configuration.

With today's plethora of architectures, GPUs and CPUs, both single and multicores, the notion of code acceleration becomes increasing nebulous. There are different approaches that could be used to define speedup. Compare a single GPU implementation against a single CPU implementation is one approach. But is the CPU implementation efficient? How fast is the CPU? Both these questions are difficult to answer. If one must code an efficient CPU implementation for every GPU code, the time lost becomes hard to justify. What if a CPU is multi-core? Should one measure speedup relative to a baseline multi-core implementation? The combinatorics quickly becomes daunting. When running on multiple GPUs, the situation is even more complex. Should one compare a parallel GPU implementation against a parallel CPU implementation? Some GPU algorithms do not have a CPU counterpart. We believe that like CPU frequency, the notion of speedup requires a careful evaluation if it is used at all. For example, proper evidence should be presented that the CPU code is sufficiently optimized. After all, the GPU code was probably constructed through careful tuning to ensure the best use of resources. Unless the same is done on the CPU, the measured speedup will probably be artificially high. A thorough description of hardware, the compiler options, and algorithmic choices is also essential. Finally, the investment of manpower, power consumption, and other indirect costs can also become non-negligible when seeking to present new implementations in the best light, and should, when possible, be taken into account.

For this study, we use CUDA version 2.3 and the following two compilers and compilation options:

Intel icc version 11.0: -O3 -x SSE4.2 -ftz -funroll-loops -unroll5

nvcc version CUDA_v2_3: -arch sm_13 -O3

We chose Intel icc over GNU gcc because it turns out that the former leads to significantly faster code for our application; note that this implies that measured GPU speedup would be significantly higher if using GNU gcc, which again shows how sensitive speedup values are to several factors, and thus how cautiously they should be interpreted. Floating point trapping is turned off (using -ftz) because underflow trapping occurs very often in the initial time steps of many seismic propagation algorithms, which can lead to severe slowdown.

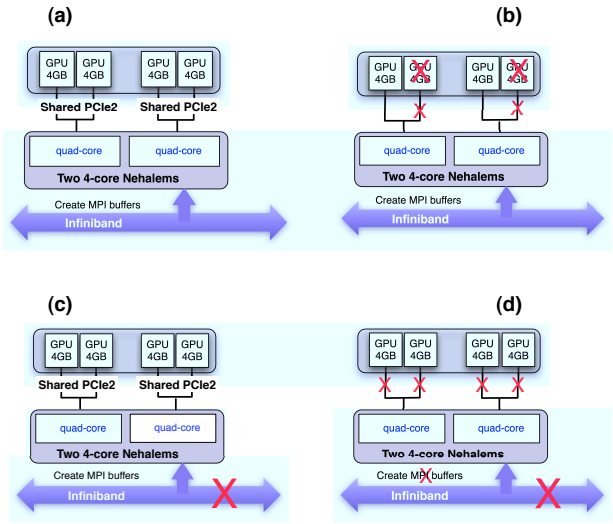

Figure 10: Description of the four types of tests performed on the cluster of 48 Teslas S1070 depicted in Figure 9.

In the case presented in this article, the CPU reference code to compute speedup is already heavily optimized [13, 38] using the ParaVer performance analysis tool [86], in particular to minimize cache misses. The current code 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 [13] and that was among the six finalists again at the SuperComputing'2008 conference for a calculation first on 62,000 processor cores and later on close to 150,000 processor cores with a sustained performance level of 0.20 petaflops [25].

Our original seismic wave propagation application being written in Fortran95 + MPI, we rewrote the computation kernel of the original code in C + MPI to facilitate interfacing with CUDA, which is currently easier from C. In a previous study [71] we found that this only has a small impact on performance, slowing down the application by about 12%.

Each mesh slice contains 446,080 spectral elements; each spectral element contains 125 grid points, many of which are shared with neighbors as explained in previous sections and in Figure 2. Each of the 192 mesh slices contains 29,606,949 unique grid points (i.e., 88,820,847 degrees of freedom since we solve for the three components of the acceleration vector). The total number of spectral elements in the mesh is 85.6 million, the total number of unique grid points is 5.684 billion, and at each time step we thus compute a total of 17 billion degrees of freedom.

Out of the 446,080 spectral elements of each mesh slice, 122,068 are 'outer' elements, i.e., elements in contact with MPI cut planes through at least one mesh point (see Figure 3) and 446,080 - 122,068 = 324,012 elements are 'inner' elements. Thus the inner and outer elements represent 27.4% and 72.6% of the total number of elements, respectively. Because we exchange 2D cut planes, composed of only one face, one edge or or one point of each element of the cut plane, the amount of data sent to other GPUs via MPI is relatively small compared to the 3D volume of each mesh slice. In practice we need to exchange values for 3,788,532 cut plane points out of the 29,606,949 grid points of each mesh slice, i.e., 12.8%, which corresponds to 43 megabytes because for each of these points we need to transfer the three components of the acceleration vector, i.e., three floats. Thus, the transfer to the CPU and from there to other GPUs can be effectively overlapped with the computation of the 3D volume of the inner elements.

6.1. Performance of the CUDA + MPI code

Let us first analyze weak scaling, i.e., how performance varies when the number of calculations to perform on each node is kept constant and the number of nodes is increased. For wave propagation applications it is often not very interesting to study strong scaling, i.e., how performance varies when the number of calculations to perform on each node is decreased linearly, because people running such large-scale HPC calculations are very often interested in using the full capacity of each node of the machine, i.e., typically around 90% of its memory capacity. We therefore designed a mesh that consumes 3.6 GB out of a maximum of 4 GB available on each GPU. The 192 mesh slices thus require a total of 700 GB of GPU memory.

All the measurements correspond to the duration (i.e., elapsed time) of 1000 time steps, keeping in mind that at each iteration the spectral-element algorithm consists of the exact same numerical operations. To

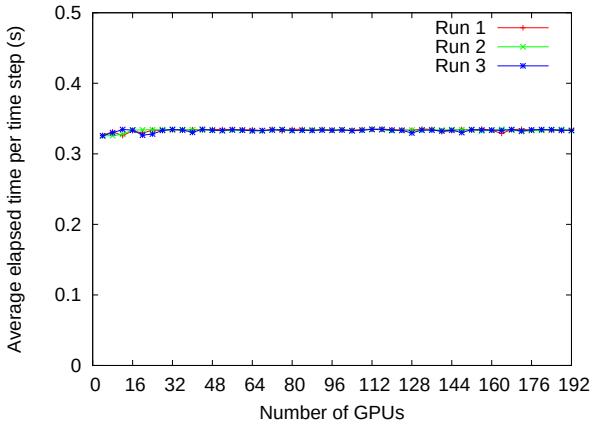

Figure 11: Average elapsed time per time step of the SEM algorithm for simulations using between 4 and 192 GPUs (i.e., the whole machine), in increments of four GPUs; each PCIe-2 bus is shared by two GPUs. We perform three runs in order to ensure reliable measurements and to measure fluctuations. The weak scaling is excellent.

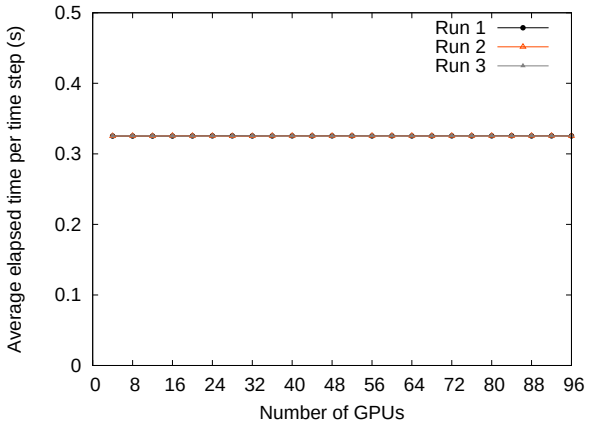

Figure 12: Average elapsed time per time step of the SEM algorithm for simulations performed using a single GPU per node (i.e., no PCIe bus sharing). The maximum number of GPUs is now 192/2 = 96. We perform three runs for each case. The fluctuations are no longer visible because they are smaller than 0.1%, and the weak scaling is now perfect.

get accurate measurements, not subject to outside interference, the nodes that participate in a particular run are not shared with other users, and each run is executed three times to ensure that the timings are reliable, and to determine any fluctuations.

In this section we perform the four types of tests described in Figure 10. Let us start with Figure 10(a), in which we perform simulations using between 4 and 192 GPUs (i.e., the whole machine), by steps of four GPUs, and each PCIe-2 bus is shared by two GPUs. The average elapsed wall-clock time per time step of the SEM algorithm is shown in Figure 11. As expected from our overlap of communications with computations, the weak scaling obtained is very good; communication is essentially free. There are small fluctuations, on the order of 2%, both between values obtained for a different number of GPUs and between repetitions of the same run.

In Figure 12 we again show the average elapsed time per time step of the SEM algorithm, but for simulations that use only one GPU per half node, thus in which the PCIe bus is not shared (Figure 10(b)). Consequently we can only use a maximum of 96 GPUs instead of 192 even when we use all the nodes of the machine. Weak scaling is now perfect, there are no fluctuations between values obtained for a different number of GPUs or between the three occurrences of the same run. This conclusively demonstrates that the small fluctuations come from sharing the PCIe bus.

To get a more quantitative estimate of the effect of PCIe bus sharing, we overlay the results of Figures 11 and 12 in Figure 13, with an expanded vertical scale, which shows that these fluctuations are small and that therefore PCIe sharing is not an issue in practice, at least in this implementation. A rough estimate of the maximum relative fluctuation amplitude between the two cases (at 60 GPUs) is 0.334 s / 0.325 s = 1.028 = 2.8%.

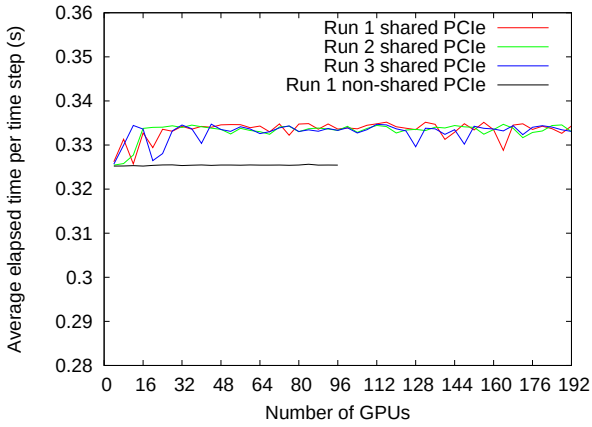

Figure 13: Comparison of the results of Figure 11 and of Figure 12 with a close-up on the vertical scale, showing that the fluctuations observed owing to sharing the PCIe bus are small, and that therefore PCIe sharing is not an issue in practice. A rough estimate of the maximum difference that can be measured on the curves (for instance for around 60 GPUs) is 0.334 / 0.325 = 1.028 = 2.8%.

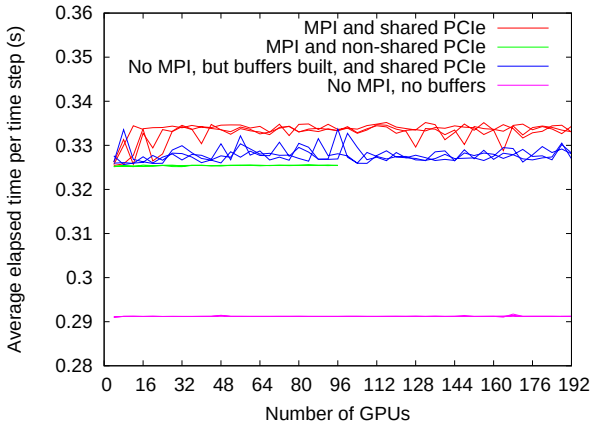

Figure 14: Comparison of four sets of measurements (with three runs per case): Average elapsed time per time step of the SEM algorithm by steps of 4 GPUs with each PCIe-2 bus shared by two GPUs (red), and with no PCIe-2 bus sharing (green); shared PCIe-2 bus with the MPI send/receives disabled but including the creation of the MPI buffers (blue); and a shared PCIe-2 bus with no MPI buffer creation or send/receives (magenta). A comparison between red and blue curves shows that at worst, the difference is around 2.8% (at 120 GPUs), indicating excellent overlap between communication and computations. Comparing the green and magenta curves gives an estimate of the total cost of running the problem on a cluster, i.e., building MPI buffers, sending/receiving them with MPI, and processing them once they are received. This cost is on the order of 12%.

We can also analyze the degree to which communications are overlapped by calculations. Figure 14 compares four sets of measurements (with three runs in each case, to make sure the measurements are reliable): the two curves of Figure 13, a calculation in which we create, copy and process the MPI buffers normally but replace the MPI send/receives by simply setting the buffers to zero (Figure 10(c)), and a calculation in which we completely turn off both the MPI communications and the creation, copying and processing of the MPI buffers (Figure 10(d)). These last two calculations produce incorrect seismic wave propagation results due to incorrect summation of the different mesh slices; however they use an identical number of operations inside each mesh slice and can therefore be used as a comparison point to estimate the cost of creating and processing the buffers and also of receiving them, i.e., the time spent in MPI_WAIT() to ensure that all the messages have been received. Note that copying the MPI buffers involves PCIe bus transfers from each GPU to the host, as explained in Section 4.6.

A comparison between the red curves (full calculation for the real problem with sharing of the PCIe bus) and the blue curve (modified calculation in which the MPI buffers are built and thus the communication costs between GPU and CPU are taken into account but the MPI send/receives are disabled) shows a good estimate of the time spent waiting for communications, i.e., is a good illustration of how/if we effectively overlap communications with calculations. In several cases (e.g., for some runs around 96 or 100 GPUs) the values are very close; communications are almost totally overlapped. In the worst cases we can estimate

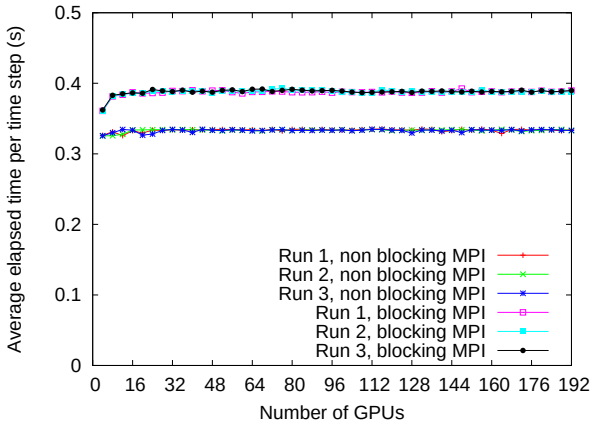

Figure 15: GPU weak scaling results of Figure 11 compared with the same simulations performed using blocking MPI. The average value of elapsed time per time step is 0.333 s in the case of non blocking MPI and 0.388 s in the case of blocking MPI, i.e., a difference of (0.388-0.333) s / 0.333 s = 0.165 = 16.5%, which shows that overlapping is efficient in our implementation of the spectral-element algorithm. The first measurement point, running on four GPUs, has a lower value in the case of blocking MPI because the four GPUs are located on the same node (see Figure 9) and thus MPI emulates blocking calls with shared memory functions without using the interconnect, resulting in significantly faster calls.

(for instance around 120 GPUs) that the difference is of the order of 0.335 s / 0.326 s = 1.028 = 2.8%: Communications are still very well overlapped with computations.

We can confirm this by estimating the cost of running the code without overlap between communication and computation using blocked MPI send and receives. In Figure 15, in which we compare the GPU weak scaling results of Figure 11 with the same simulations performed using blocking MPI, we see that the average value of elapsed time per time step is 0.333 s in the case of non blocking MPI and 0.388 s in the case of blocking MPI, i.e., a difference of (0.388-0.333) s / 0.333 s = 0.165 = 16.5%. This shows that overlapping is efficient in our implementation of the spectral-element algorithm and that using a simpler blocking implementation would have resulted in a loss of performance. More importantly, previous experiments have shown blocking communications to be untenable because of bottlenecks once the number of processors exceeds on the order of 2,000 [38], resulting in very poor scaling above that limit. Although the number of GPUs in this article is far below this threshold, we anticipate clusters with several thousand GPUs in the near future and thus using blocking MPI is not a valid option.

Comparing the green curves (full calculation for the real problem without sharing of the PCIe bus) and the magenta curves (modified calculation in which the MPI buffers are not built, nothing gets copied through the PCIe bus, and MPI is completely turned off) gives an estimate of the total cost of running the problem on a cluster, i.e., having cut the mesh into pieces, which implies building MPI buffers, sending/receiving them with MPI, and processing them once they are received. We measure that this total cost is of the order of 0.325 s / 0.291 s = 1.117 = 11.7%. We emphasize that this cost does not affect the efficiency and scalability of the code, as it is not serialized.

Figure 16 shows some reference CPU weak scaling results, using four cores per node out of the eight cores that are available, for the same mesh as for the GPU tests, i.e., with 3.6 GB on each core, or 14.4 GB per node, out of a maximum of 24 GB per node. To use the resources in a balanced way, we assign two MPI processes to each of the two quad-core Nehalem processors on each node. We pin processes to cores using process binding to avoid process migration by the Linux kernel during the runs. Again, each run is repeated three times. We overlay reference CPU weak scaling results using all eight cores per node. To do this, we cut each mesh slice in two but use twice as many MPI processes, thus using 1.8 GB of memory per core instead of 3.6 GB but keeping the same total mesh size. On average the weak scaling is excellent because communications are almost completely overlapped by calculations and thus hidden, as in the GPU cases discussed above. We observe small fluctuations, with a maximum amplitude of about 2%, that are larger when using the whole 8 cores per node and that are probably due to resource sharing between processes, to NUMA effects, to interconnect contention or to memory bus contention. The average elapsed time per time step is 6.61 seconds when we use four cores and 4.08 seconds when we use eight cores. Therefore by using eight cores instead of four, as expected due to resource sharing, we only gain a factor of 6.61 s / 4.08 s = 1.62, significantly lower than the ideal factor of 2.

Figure 17 shows GPU/CPU speedup, computed as the ratio of the average timings of the reference CPU runs from Figure 16 and the average timings of the GPU runs of Figure 11 (with MPI + PCIe bus sharing).

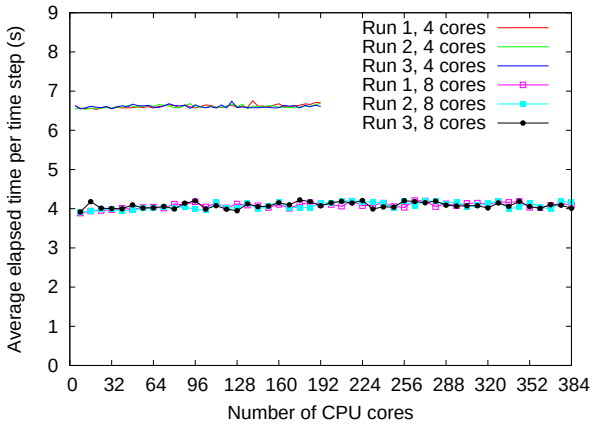

Figure 16: Solid lines: Reference CPU weak scaling results, using four cores per node out of the eight cores that are available, with 3.6 GB per core out of a maximum of 24 GB per node. We assign two MPI processes to each of the two quad-core Nehalem processors on each node. Lines with symbols: Reference CPU weak scaling results for the same mesh but decomposed into twice more slices, i.e., using all eight cores that are available per node, with 1.8 GB per core out of 24 GB. Weak scaling is excellent in both cases owing to very good overlap between communication and calculation (as illustrated in Figure 15). Fluctuations are more pronounced when all eight cores are used per node because of resource sharing.

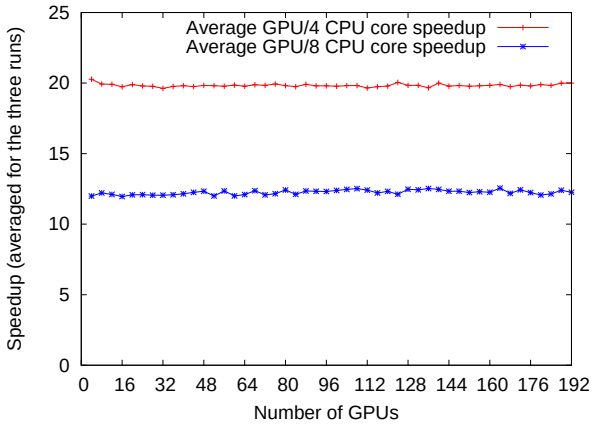

Figure 17: GPU/CPU speedup obtained, computed by averaging the results of the three CPU runs of Figure 16 and dividing them by the average of the three GPU runs of Figure 11. The average speedup of GPU runs versus CPU runs on four cores per node (i.e., two cores per half node) (red symbols) for the 48 measurements is 19.83, the maximum is 20.26 and the minimum is 19.62; versus CPU runs on all eight cores per node (blue symbols), the average is 12.25, the maximum is 12.56 and the minimum is 11.96.

The average speedup of GPU runs versus CPU runs on four cores per node is 19.83, the maximum is 20.26 and the minimum is 19.62; versus CPU runs on eight cores per node, the average is 12.25, the maximum is 12.56 and the minimum is 11.96. The values around 20x are smaller than the factor of 25x that we got in our first article [71], which only considered the case of a single GPU (and no MPI). The two reasons for that are that the Intel Nehalem processors used here as a reference are faster than the processors used in [71], and that the Teslas S1070 that we use in this study have less memory bandwidth, $\simeq$100 GB/s external bandwidth compared to $\simeq$140 GB/s in the GeForce GTX 280 card used in [71].

6.2. Optimizations considered

Out of the 11.7% overhead associated to the parallel implementation of SPECFEM3D on a cluster of GPUs, on the order of 2.8% is solely due to the time spent waiting for a few remaining non-blocking MPI receives. Thus, less than 10% of the total time is associated with building and processing the MPI buffers that transfer boundary data of slices between the GPUs. These buffers contain the acceleration data on the four surface planes of each slice (also called cut planes). They are filled and emptied by specialized kernels responsible for both extracting and merging the cut planes from/to the 3D local acceleration arrays. Data from the four cut planes are merged into a single array to minimize the number of individual transfers between host and device through the PCIe bus. Although the buffer must again be broken up on the CPU

to communicate the data to four different slices via MPI non-blocking sends, this approach is more efficient than transferring four individual cut planes via PCIe.

The time to transfer these buffers between the host and the device, included in the 10% overhead, could, in principle be partly overlapped with calculations using the CUDA "stream" or "zero copy" mechanisms. Streams are a mechanism of asynchronous data transfer, which facilitates overlap of kernel and CPU computation, and device/host transfers. Streams and kernels have an associated stream ID. Kernels and data transfers via streams can be overlapped automatically by the hardware/driver if their IDs are different. Assuming N streams, we can theoretically overlap $N-1$ transfers out of N, notwithstanding the additional overhead cost of the N different transfers. We performed tests that yielded only diminishing returns compared to the block transfer of the entire cut plane data. Therefore, the additional complexity of the code is not justified.

Zero copy, on the other hand, is a mechanism available since CUDA 2.2, which enables transparent data access from host memory as if it were already stored on the device. The potential disadvantages of this approach are three-fold: first, the use of page-locked memory on the CPU, which then becomes unavailable for other functions; second, transfers go through the PCIe-2 bus, and are necessarily slow compared to access to device memory without leaving the GPU; finally, the user has no control over how data transfer is implemented by the system. Given the small potential return (if any) on coding investment, we decided not to implement streams or zero copy mechanisms in our code.

Data transfers to and from device memory on the GPU are most efficient when coalesced. If the access pattern of data does not satisfy the constraints for maximum performance [8], yet have spatial locality, the texture hardware offers a good alternative. Using textures, memory requests are routed through a texture cache, thus accelerating transfers of data that are already cached. Only data loads are possible because there is no cache coherency. We have some uncoalesced reads in kernel 2 (cf. Figure 5) because of the indirect addressing when reading from the global displacement and acceleration vectors at the level of a given spectral element. We therefore implement these reads, through texture binding, to the displacement and acceleration arrays. Tests indicate that the total elapsed time spent per time step decreases by only one to two percent. We nonetheless keep the texture binding due to its straightforward implementation.

7. Conclusions and future work

We have investigated how to combine GPU CUDA programming with non-blocking message passing based on MPI on a cluster of Tesla GPUs to accelerate a high-order finite-element software package that performs the numerical simulation of seismic wave propagation. Such an application can be useful, for instance, to model seismic waves resulting from earthquakes or from active seismic experiments in the oil industry.

This application executes accurately in single precision. Hence, current GPU hardware, which is significantly faster for single precision arithmetic, is suitable and sufficient.

We validated the algorithm by performing a comparison between the CUDA + MPI version and the original C + MPI version of the code running on CPUs only. We subsequently modeled a real earthquake that occurred in Bolivia in 1994 and obtained the same physical behavior as in real seismic data recorded during the event and available in the geophysical literature. We also obtained an excellent fit with an independent quasi-analytical solution computed based upon normal-mode summation.

We then performed several numerical tests to compare performance between the GPU + MPI version and the original C + MPI version without CUDA and showed that we obtain a speedup of 20x or 12x, depending on how the problem is mapped to the reference CPU cluster.

In future work we would like to investigate using OpenCL [11, 4] instead of CUDA to make the code portable to non-NVIDIA hardware, including multi-core systems. We also plan to examine any potential advantages of the multiple-kernel execution capabilities of the FERMI chip. We could also investigate how to handle more complex meshes with a non regular domain decomposition topology, e.g., meshes decomposed based on a domain-decomposition library such as SCOTCH [87] or METIS [88].

Acknowledgments

The authors would like to thank Jean Roman, Jean-François Méhaut, Christophe Merlet, Matthieu Ospici, Xavier Vigouroux, Philippe Thierry and Roland Martin for fruitful discussion about GPU computing. The calculations were performed on the 'Titane' BULL Novascale R422 GPU cluster at CCRT/CEA/GENCI in Bruyères-le-Châtel, France, with support from Stéphane Requena, Christine Ménaché, Édouard Audit, Jean-Noël Richet, Gilles Wiber, Julien Derouillat, Laurent Nguyen and Pierre Bonneau. The comments of

two anonymous reviewers, the Associate Editor, Basil Nyaku and Alvin Bayliss improved the manuscript. This research was funded in part by French ANR grant NUMASIS ANR-05-CIGC-002, by French CNRS, INRIA and IUF, by German Deutsche Forschungsgemeinschaft projects TU102/22-1 and TU102/22-2, and by German BMBF in the SKALB project 01IH08003D of call 'HPC Software für skalierbare Parallelrechner'.

References

[1] J. D. Owens, M. Houston, D. P. Luebke, S. Green, J. E. Stone, J. C. Phillips, GPU Computing, Proceedings of the IEEE 96 (5) (2008) 879–899, doi:10.1109/JPROC.2008.917757.

[2] M. Garland, S. L. Grand, J. Nickolls, J. A. Anderson, J. Hardwick, S. Morton, E. H. Phillips, Y. Zhang, V. Volkov, Parallel Computing Experiences with CUDA, IEEE Micro 28 (4) (2008) 13–27, doi:10.1109/MM.2008.57.

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, K. Skadron, A performance study of general-purpose applications on graphics processors using CUDA, Journal of Parallel and Distributed Computing 68 (10) (2008) 1370–1380, doi:10.1016/j.jpdc.2008.05.014.

[4] D. B. Kirk, W.-m. W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, Morgan Kaufmann, Boston, Massachusetts, USA, 2010.

[5] NVIDIA Corporation, NVIDIA's Next Generation CUDA Compute Architecture: FERMI, Tech. Rep., NVIDIA, Santa Clara, California, USA, URL `http://www.nvidia.com/object/fermi_architecture.html`, 22 pages, 2009.

[6] D. Göddeke, Fast and Accurate Finite-Element Multigrid Solvers for PDE Simulations on GPU Clusters, Ph.D. thesis, Technische Universität Dortmund, Fakultät für Mathematik, `http://hdl.handle.net/2003/27243`, 2010.

[7] J. D. Owens, D. P. Luebke, N. K. Govindaraju, M. J. Harris, J. Krüger, A. E. Lefohn, T. J. Purcell, A Survey of General-Purpose Computation on Graphics Hardware, Computer Graphics Forum 26 (1) (2007) 80–113, doi:10.1111/j.1467-8659.2007.01012.x.

[8] NVIDIA Corporation, NVIDIA CUDA Programming Guide version 2.3, Santa Clara, California, USA, URL `http://www.nvidia.com/cuda`, 139 pages, 2009.

[9] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym, NVIDIA Tesla: A Unified Graphics and Computing Architecture, IEEE Micro 28 (2) (2008) 39–55, doi:10.1109/MM.2008.31.

[10] J. Nickolls, I. Buck, M. Garland, K. Skadron, Scalable Parallel Programming with CUDA, ACM Queue 6 (2) (2008) 40–53, doi:10.1145/1365490.1365500.

[11] Khronos OpenCL Working Group, The OpenCL Specification, Version 1.0, `http://www.khronos.org/opencl`, 2008.

[12] K. Fatahalian, M. Houston, A Closer Look at GPUs, Communications of the ACM 51 (10) (2008) 50–57, doi:10.1145/1400181.1400197.

[13] 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, Proceedings of the ACM/IEEE Supercomputing SC'2003 conference (2003) 4–11doi:10.1109/SC.2003.10023, Gordon Bell Prize winner article.

[14] Q. Liu, J. Polet, D. Komatitsch, J. Tromp, Spectral-element moment tensor inversions for earthquakes in Southern California, Bull. Seismol. Soc. Am. 94 (5) (2004) 1748–1761, doi:10.1785/012004038.

[15] E. Chaljub, D. Komatitsch, J. P. Vilotte, Y. Capdeville, B. Valette, G. Festa, Spectral Element Analysis in Seismology, in: R.-S. Wu, V. Maupin (Eds.), Advances in wave propagation in heterogeneous media, vol. 48 of *Advances in Geophysics*, Elsevier - Academic Press, London, UK, 365–419, 2007.

[16] J. Tromp, D. Komatitsch, Q. Liu, Spectral-Element and Adjoint Methods in Seismology, Communications in Computational Physics 3 (1) (2008) 1–32.

[17] G. Cohen, P. Joly, N. Tordjman, Construction and analysis of higher-order finite elements with mass lumping for the wave equation, in: R. Kleinman (Ed.), Proceedings of the second international conference on mathematical and numerical aspects of wave propagation, SIAM, Philadelphia, Pennsylvania, USA, 152–160, 1993.

[18] E. Priolo, J. M. Carcione, G. Seriani, Numerical simulation of interface waves by high-order spectral modeling techniques, J. Acoust. Soc. Am. 95 (2) (1994) 681–693.

[19] E. Faccioli, F. Maggio, R. Paolucci, A. Quarteroni, 2D and 3D elastic wave propagation by a pseudo-spectral domain decomposition method, J. Seismol. 1 (1997) 237–251.

[20] M. O. Deville, P. F. Fischer, E. H. Mund, High-Order Methods for Incompressible Fluid Flow, Cambridge University Press, Cambridge, United Kingdom, 2002.

[21] E. Chaljub, Y. Capdeville, J. P. Vilotte, Solving Elastodynamics in a Fluid-Solid Heterogeneous Sphere: a Parallel Spectral-Element Approximation on Non-Conforming Grids, J. Comput. Phys. 187 (2) (2003) 457–491.

[22] J. D. De Basabe, M. K. Sen, Grid dispersion and stability criteria of some common finite-element methods for acoustic and elastic wave equations, Geophysics 72 (6) (2007) T81–T95, doi:10.1190/1.2785046.

[23] G. Seriani, S. P. Oliveira, Dispersion analysis of spectral-element methods for elastic wave propagation, Wave Motion 45 (2008) 729–744, doi:10.1016/j.wavemoti.2007.11.007.

[24] P. E. J. Vos, S. J. Sherwin, R. M. Kirby, From h to p efficiently: Implementing finite and spectral/hp element methods to achieve optimal performance for low- and high-order discretisations, J. Comput. Phys. 229 (2010) 5161–5181, doi:10.1016/j.jcp.2010.03.031.

[25] L. Carrington, D. Komatitsch, M. Laurenzano, M. Tikir, D. Michéa, N. Le Goff, A. Snavely, J. Tromp, High-frequency simulations of global seismic wave propagation using SPECFEM3D_GLOBE on 62 thousand processor cores, Proceedings of the ACM/IEEE Supercomputing SC'2008 conference (2008) 1–11doi:10.1145/1413370.1413432, article #60, Gordon Bell Prize finalist article.

[26] R. Martin, D. Komatitsch, C. Blitz, N. Le Goff, Simulation of seismic wave propagation in an asteroid based upon an unstructured MPI spectral-element method: blocking and non-blocking communication strategies, Lecture Notes in Computer Science 5336 (2008) 350–363.

[27] S. J. Sherwin, G. E. Karniadakis, A triangular spectral element method: applications to the incompressible Navier-Stokes equations, Comput. Meth. Appl. Mech. Eng. 123 (1995) 189–229.

[28] M. A. Taylor, B. A. Wingate, A generalized diagonal mass matrix spectral element method for non-quadrilateral elements, Appl. Num. Math. 33 (2000) 259–265.

[29] D. Komatitsch, R. Martin, J. Tromp, M. A. Taylor, B. A. Wingate, Wave propagation in 2-D elastic media using a spectral element method with triangles and quadrangles, J. Comput. Acoust. 9 (2) (2001) 703–718, doi:10.1142/S0218396X01000796.

[30] E. D. Mercerat, J. P. Vilotte, F. J. Sánchez-Sesma, Triangular spectral-element simulation of two-dimensional elastic wave propagation using unstructured triangular grids, Geophys. J. Int. 166 (2) (2006) 679–698.

[31] R. S. Falk, G. R. Richter, Explicit Finite Element Methods for Symmetric Hyperbolic Equations, SIAM J. Numer. Anal. 36 (3) (1999) 935–952, doi:10.1137/S0036142997329463.

[32] F. Q. Hu, M. Y. Hussaini, P. Rasetarinera, An analysis of the discontinuous Galerkin method for wave propagation problems, J. Comput. Phys. 151 (2) (1999) 921–946, doi:10.1006/jcph.1999.6227.

[33] B. Rivière, M. F. Wheeler, Discontinuous Finite Element Methods for Acoustic and Elastic Wave Problems, Contemporary Mathematics 329 (2003) 271–282.

[34] P. Monk, G. R. Richter, A Discontinuous Galerkin Method for Linear Symmetric Hyperbolic Systems in Inhomogeneous Media, Journal of Scientific Computing 22-23 (1-3) (2005) 443–477, doi:10.1007/s10915-004-4132-5.

[35] M. J. Grote, A. Schneebeli, D. Schötzau, Discontinuous Galerkin Finite Element Method for the Wave Equation, SIAM Journal on Numerical Analysis 44 (6) (2006) 2408–2431, doi:10.1137/05063194X.

[36] M. Bernacki, S. Lanteri, S. Piperno, Time-Domain Parallel Simulation Of Heterogeneous Wave Propagation On Unstructured Grids Using Explicit, Nondiffusive, Discontinuous Galerkin Methods, J. Comput. Acoust. 14 (1) (2006) 57–81.

[37] M. Dumbser, M. Käser, E. Toro, An arbitrary high-order discontinuous Galerkin method for elastic waves on unstructured meshes, Part V: Local time stepping and p-adaptivity, Geophys. J. Int. 171 (2) (2007) 695–717, doi:10.1111/j.1365-246X.2007.03427.x.

[38] 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, Lecture Notes in Computer Science 5336 (2008) 364–377.

[39] V. Volkov, J. W. Demmel, Benchmarking GPUs to tune dense linear algebra, in: SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, 1–11, doi:10.1145/1413370.1413402, 2008.

[40] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, S. Tomov, Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects, Journal of Physics: Conference Series 180 (2009) 012037, doi:10.1088/1742-6596/180/1/012037.

[41] P. Micikevicius, 3D finite-difference computation on GPUs using CUDA, in: GPGPU-2: Proceedings of the 2nd Workshop on General Purpose Processing on Graphics Processing Units, Washington, DC, USA, 79–84, doi:10.1145/1513895.1513905, 2009.

[42] N. Bell, M. Garland, Implementing sparse matrix-vector multiplication on throughput-oriented processors, in: SC'09: Proceedings of the 2009 ACM/IEEE conference on Supercomputing, ACM, New York, USA, 1–11, doi:10.1145/1654059.1654078, 2009.

[43] A. Corrigan, F. Camelli, R. Löhner, J. Wallin, Running unstructured grid based CFD solvers on modern graphics hardware, in: 19th AIAA Computational Fluid Dynamics Conference, 1–11, aIAA 2009-4001, 2009.

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

[45] R. Abdelkhalek, H. Calandra, O. Coulaud, J. Roman, G. Latu, Fast Seismic Modeling and Reverse Time Migration on a GPU Cluster, in: W. W. Smari, J. P. McIntire (Eds.), High Performance Computing & Simulation 2009, Leipzig, Germany, 36–44, http://hal.inria.fr/docs/00/40/39/33/PDF/hpcs.pdf, 2009.

[46] D. Michéa, D. Komatitsch, Accelerating a 3D finite-difference wave propagation code using GPU graphics cards, Geophys. J. Int. 182 (1) (2010) 389–402, doi:10.1111/j.1365-246X.2010.04616.x.

[47] A. Klöckner, T. Warburton, J. Bridge, J. S. Hesthaven, Nodal discontinuous Galerkin methods on graphics processors, J. Comput. Phys. 228 (2009) 7863–7882, doi:10.1016/j.jcp.2009.06.041.

[48] S. Chaillat, M. Bonnet, J.-F. Semblat, A multi-level fast multipole BEM for 3-D elastodynamics in the frequency domain, Comput. Meth. Appl. Mech. Eng. 197 (49-50) (2008) 4233–4249, doi:10.1016/j.cma.2008.04.024.

[49] N. A. Gumerov, R. Duraiswami, Fast multipole methods on graphics processors, J. Comput. Phys. 227 (2008) 8290–8313, doi:10.1016/j.jcp.2008.05.023.

[50] N. Raghuvanshi, R. Narain, M. C. Lin, Efficient and Accurate Sound Propagation Using Adaptive Rectangular Decomposition, IEEE Transactions on Visualization and Computer Graphics 15 (5) (2009) 789–801, doi:10.1109/TVCG.2009.28.

[51] W. Wu, P. A. Heng, A hybrid condensed finite element model with GPU acceleration for interactive 3D soft tissue cutting: Research Articles, Computer Animation and Virtual Worlds archive 15 (3-4) (2004) 219–227, doi:10.1002/cav.v15:3/4.

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

[53] 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, 1–4, doi:10.1109/ISAPE.2006.353223, 2006.

[54] Z. A. Taylor, M. Cheng, S. Ourselin, High-Speed Nonlinear Finite Element Analysis for Surgical Simulation Using Graphics Processing Units, IEEE Transactions on Medical Imaging 27 (5) (2008) 650–663, doi:10.1109/TMI.2007.913112.

[55] Z. Fan, F. Qiu, A. E. Kaufman, S. Yoakum-Stover, GPU Cluster for High Performance Computing, in: SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing, 47, doi: 10.1109/SC.2004.26, 2004.

[56] 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 Computing 33 (10-11) (2007) 685–699.

[57] D. Göddeke, H. Wobker, R. Strzodka, J. Mohd-Yusof, P. S. McCormick, S. Turek, Co-Processor Acceleration of an Unmodified Parallel Solid Mechanics Code with FEASTGPU, International Journal of Computational Science and Engineering 4 (4) (2009) 254–269.

[58] D. Göddeke, S. H. Buijssen, H. Wobker, S. Turek, GPU Acceleration of an Unmodified Parallel Finite Element Navier-Stokes Solver, in: W. W. Smari, J. P. McIntire (Eds.), High Performance Computing & Simulation 2009, Leipzig, Germany, 12–21, 2009.

[59] M. Fatica, Accelerating Linpack with CUDA on heterogenous clusters, in: D. Kaeli, M. Leeser (Eds.), GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, no. 383 in ACM International Conference Proceeding Series, 46–51, doi:10.1145/1513895.1513901, 2009.

[60] J. C. Phillips, J. E. Stone, K. Schulten, Adapting a message-driven parallel application to GPU-accelerated clusters, in: SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, 1–9, doi:10.1145/1413370.1413379, 2008.

[61] 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, doi: 10.1016/j.jcp.2008.01.047.

[62] J. C. Thibault, I. Senocak, CUDA Implementation of a Navier-Stokes Solver on Multi-GPU Desktop Platforms for Incompressible Flows, in: Proceedings of the 47th AIAA Aerospace Sciences Meeting, 1–15, 2009.

[63] E. H. Phillips, Y. Zhang, R. L. Davis, J. D. Owens, Rapid Aerodynamic Performance Prediction on a Cluster of Graphics Processing Units, in: Proceedings of the 47th AIAA Aerospace Sciences Meeting, 1–11, 2009.

[64] J. A. Stuart, J. D. Owens, Message Passing on Data-Parallel Architectures, in: Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium, 1–12, doi: 10.1109/IPDPS.2009.5161065, 2009.

[65] V. V. Kindratenko, J. J. Enos, G. Shi, M. T. Showerman, G. W. Arnold, J. E. Stone, J. C. Phillips, W. Hwu, GPU Clusters for High-Performance Computing, in: Proceedings on the IEEE Cluster'2009 Workshop on Parallel Programming on Accelerator Clusters (PPAC'09), New Orleans, Louisiana, USA, 1–8, 2009.

[66] Z. Fan, F. Qiu, A. E. Kaufman, Zippy: A Framework for Computation and Visualization on a GPU Cluster, in: G. Drettakis, R. Scopigno (Eds.), Proceedings of the Eurographics'2008 Symposium on Parallel Graphics and Visualization (EGPGV'08), vol. 27(2), Hersonissos, Crete, Greece, 341–350, 2008.

[67] M. Strengert, C. Müller, C. Dachsbacher, T. Ertl, CUDASA: Compute Unified Device And Systems Architecture, in: J. Favre, K. L. Ma, D. Weiskopf (Eds.), Proceedings of the Eurographics'2008 Symposium on Parallel Graphics and Visualization (EGPGV'08), Hersonissos, Crete, Greece, 49–56, 2008.

[68] D. Göddeke, R. Strzodka, S. Turek, Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations, International Journal of Parallel, Emergent and Distributed Systems 22 (4) (2007) 221–256.

[69] D. Komatitsch, J. Tromp, Introduction to the spectral-element method for 3-D seismic wave propagation, Geophys. J. Int. 139 (3) (1999) 806–822, doi:10.1046/j.1365-246x.1999.00967.x.

[70] D. Komatitsch, J. Tromp, Spectral-Element Simulations of Global Seismic Wave Propagation-I. Validation, Geophys. J. Int. 149 (2) (2002) 390–412, doi:10.1046/j.1365-246X.2002.01653.x.

[71] D. Komatitsch, D. Michéa, G. Erlebacher, Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA, Journal of Parallel and Distributed Computing 69 (5) (2009) 451–460, doi:10.1016/j.jpdc.2009.01.006.

[72] K. van Wijk, D. Komatitsch, J. A. Scales, J. Tromp, Analysis of strong scattering at the micro-scale, J. Acoust. Soc. Am. 115 (3) (2004) 1006–1011, doi:10.1121/1.1647480.

[73] G. Seriani, E. Priolo, A spectral element method for acoustic wave simulation in heterogeneous media, Finite Elements in Analysis and Design 16 (1994) 337–348.

[74] C. Canuto, M. Y. Hussaini, A. Quarteroni, T. A. Zang, Spectral methods in fluid dynamics, Springer-Verlag, New-York, USA, 1988.

[75] T. J. R. Hughes, The finite element method, linear static and dynamic finite element analysis, Prentice-Hall International, Englewood Cliffs, New Jersey, USA, 1987.

[76] T. Nissen-Meyer, A. Fournier, F. A. Dahlen, A 2-D spectral-element method for computing spherical-earth seismograms - II. Waves in solid-fluid media, Geophys. J. Int. 174 (2008) 873–888, doi: 10.1111/j.1365-246X.2008.03813.x.

[77] J. D. De Basabe, M. K. Sen, Stability of the high-order finite elements for acoustic or elastic wave propagation with high-order time stepping, Geophys. J. Int. 181 (1) (2010) 577–590, doi:10.1111/j.1365-246X.2010.04536.x.

[78] K. T. Danielson, R. R. Namburu, Nonlinear dynamic finite element analysis on parallel computers using Fortran90 and MPI, Advances in Engineering Software 29 (3-6) (1998) 179–186.

[79] 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, 41–46, 1982.

[80] 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. Meth. Appl. Mech. Eng. 61 (2) (1987) 215–248.

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

[82] J.-J. Droux, An algorithm to optimally color a mesh, Comput. Meth. Appl. Mech. Eng. 104 (2) (1993) 249–260, doi:10.1016/0045-7825(93)90199-8.

[83] A. M. Dziewoński, D. L. Anderson, Preliminary reference Earth model, Phys. Earth Planet. In. 25 (1981) 297–356.

[84] W. Jiao, T. C. Wallace, S. L. Beck, Evidence for static displacements from the June 9, 1994 deep Bolivian earthquake, Geophys. Res. Lett. 22 (16) (1995) 2285–2288.

[85] G. Ekström, Calculation of static deformation following the Bolivia earthquake by summation of Earth's normal modes, Geophys. Res. Lett. 22 (16) (1995) 2289–2292.

[86] 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, 80.2, doi:10.1109/IPDPS.2003.1213183, URL `www.cepba.upc.es/paraver`, 2003.

[87] F. Pellegrini, J. Roman, SCOTCH: A Software Package for Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs, Lecture Notes in Computer Science 1067 (1996) 493–498.

[88] G. Karypis, V. Kumar, A Fast and High-Quality Multilevel Scheme for Partitioning Irregular Graphs, SIAM Journal on Scientific Computing 20 (1) (1998) 359–392.