Computer Science - Research and Development manuscript No.
(will be inserted by the editor)

Modeling the propagation of elastic waves using spectral elements on a cluster of 192 GPUs

Dimitri Komatitsch · **Dominik Göddeke** · **Gordon Erlebacher** · **David Michéa**

Received: Jan 11, 2010 / Accepted: Feb 24, 2010

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 GPU-enhanced cluster. Mesh coloring enables an efficient accumulation of degrees of freedom in the assembly process over an unstructured mesh. We use non-blocking MPI and show that computations and communications over the network and between the CPUs and the GPUs are almost fully overlapped. The GPU solver scales excellently up to 192 GPUs and achieves significant speedup over a carefully tuned equivalent CPU code.

Keywords GPU computing · finite elements · spectral elements · seismic modeling · CUDA · MPI

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 Bundesministerium für Bildung und Forschung in the SKALB project 01IH08003D of call ‘HPC Software für skalierbare Parallelrechner’.

Dimitri Komatitsch
Université de Pau, CNRS & INRIA Magique-3D,
Laboratoire de Modélisation et d’Imagerie en Géosciences UMR 5212,
E-mail: dimitri.komatitsch@univ-pau.fr

Dominik Göddeke
Institut für Angewandte Mathematik, TU Dortmund, Germany
E-mail: dominik.goeddeke@math.tu-dortmund.de

Gordon Erlebacher
Department of Scientific Computing, Florida State University, USA
E-mail: gerlebacher@fsu.edu

David Michéa
Bureau de Recherches Géologiques et Minières, Orléans, France
E-mail: davidmichea@gmail.com

1 Introduction

Over the past several years, graphics processors (GPUs) have rapidly gained interest as a viable architecture for general purpose computations. Current GPUs can be seen as wide-SIMD many-core designs, with a hardware scheduler that keeps thousands of threads ‘in flight’ simultaneously by efficiently suspending threads stalled for memory transactions. GPUs thus maximize computational and memory throughput of an entire ‘compute kernel’ (a sequence of computations that does not need synchronization in DRAM), in contrast to CPU (cores) that minimize the latency of individual operations and alleviate the memory wall problem by ever larger hierarchies of on-chip cache memory. We refer to a recent article by Fatahalian and Houston (2008) for an overview of GPU architecture and a comparison to multithreaded CPU designs. Significant speedups in the range of five to fifty have been reported for many application domains, see for instance recent surveys on the field by Owens et al (2008), Garland et al (2008) and Che et al (2008). Furthermore, GPUs are beneficial in terms of energy efficiency and other costs related to operating large-scale HPC installations, a topic that is becoming increasingly important (‘green computing’).

Related work on GPU clusters: Fan et al (2004) 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. More recently, Göddeke et al (2007) 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. A molecular dynamics framework on GPU clusters has been presented by Phillips et al (2008), and Phillips et al (2009) have accelerated an Euler solver on a 16-node GPU cluster. Finally, Kindratenko et al (2009) discuss many issues related to installing and operating large GPU clusters.

Seismic modeling and geosciences on GPUs: Micikevicius (2009) and Abdelkhalek et al (2009) have recently calculated seismic reverse time migration for the oil and gas industry on GPUs. 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. In previous work, we have used finite-difference and finite-element algorithms to model forward seismic wave propagation (Komatitsch et al (2009), Michéa and Komatitsch (2010)).

1.1 Article contribution and overview

Despite all advances and achievements, it must be kept in mind that GPUs are co-processors in the traditional sense. Several GPUs within one cluster node have to be coordinated by the CPU(s); and for parallel computations on clusters, the CPU retains full control of the interconnect. Data must be moved from device memory to host memory prior to its transmission over the network, and vice versa. Additionally, the bus to the host can be shared by several GPUs. Given the high speedups that have been reported for the serial case and on small clusters of GPUs, a satisfactory scaling is not an automatic consequence. The ratio of computation to communication changes unfavorably, and sequential and communication-intensive stages of the code may become dominant. 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. This effect can only be avoided by reducing the serial cost of CPU to CPU and CPU to GPU communication to a negligible value.

In this article, we report on our experiences in extending the geophysics software SPECFEM3D to execute on GPU clusters. We demonstrate how to overlap all additional bus transfers and the interconnect communication with computations on the device via techniques that are applicable also in other application domains, and demonstrate almost perfect weak scalability on a 192-GPU cluster.

2 Background

2.1 The spectral element method and solution algorithm

We resort to the Spectral Element Method (SEM) to simulate numerically the propagation of seismic waves resulting from earthquakes or from active seismic acquisition experiments in the oil industry (Chaljub et al (2007); Tromp et al (2008)). Another use is to simulate ultrasonic laboratory experiments (van Wijk et al (2004)). 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. It is more flexible than traditional global pseudospectral techniques (Komatitsch et al (1996)).

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

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

where $\mathbf{u}$ is the displacement vector, σ the symmetric, second-order stress tensor, ε 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,

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

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

In a SEM, the physical domain is subdivided into mesh cells within which variables are approximated by high order interpolants. 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. A Jacobian transform then defines the mapping between Cartesian points $\mathbf{x}=(x,y,z)$ within a deformed, hexahedral element Ω_e and the reference cube.

To represent the displacement field in an element, the SEM uses Lagrange polynomials of degree 4 to 10, typically, for the interpolation of functions (Seriani and Priolo, 1994; De Basabe and Sen, 2007). Chaljub et al (2007) and De Basabe and Sen (2007) find that choosing the degree $n=4$ gives a good compromise between accuracy and time step duration. The control points ξ_α are chosen to be the $n+1$ Gauss-Lobatto-Legendre (GLL) points. 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 (Chaljub et al, 2007; Tromp et al, 2008), which can be implemented efficiently on large parallel machines (e.g., Carrington et al, 2008). Functions f that represent the physical unknowns on an element are then interpolated in terms of triple products of Lagrange polynomials of degree n.

Solving the weak form of the equations of motion (2) requires numerical integrations over the elements. A GLL integration rule is used for that purpose, and 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

$$\mathrm{M}\ddot{\mathrm{U}} + \mathrm{K}\mathrm{U} = \mathrm{F} \; , \tag{3}$$

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 Chaljub et al (2007). Time integration of this system is usually performed based on a second-order centered finite-difference Newmark time scheme (e.g., Hughes (1987); Chaljub et al (2007); Tromp et al (2008)), although higher-order time schemes can be used if necessary (Nissen-Meyer et al (2008)).

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. All the time steps have identical cost because the mesh is static and the algorithm is fully explicit, which greatly facilitates optimization.

2.2 Our simulation software: SPECFEM3D

In the last decade, in collaboration with several colleagues, we developed SPECFEM3D, a software package that performs the three-dimensional numerical simulation of seismic wave propagation resulting from earthquakes or from active seismic experiments in the oil industry, based on the spectral-element method (SEM, Komatitsch et al (2003); Liu et al (2004); Chaljub et al (2007); Tromp et al (2008)). 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 2,000 to 4,000, and take two to five days of elapsed wall-clock time to complete (Komatitsch et al (2003, 2008)). The largest run that we have performed ran on close to 150,000 processor cores with a sustained performance level of 0.20 petaflops (Carrington et al (2008)).

In this article we 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.

3 Porting SPECFEM3D to GPU clusters

3.1 Meshing, partitioning and load balancing

In a preprocessing step, we mesh the region of the Earth in which the earthquake occurred with hexahedra. Because the computational costs associated with this stage are amortized over many time steps in the course of the simulation, a CPU implementation is justified, rather than also porting it to the GPU. Next, we split the mesh into slices, i.e., cone-shaped sections of the Earth from its surface to the outer core. The model of the Earth that we use is that of Dziewoński and Anderson (1981), which is classical in the seismological community. We identify slices with MPI ranks, and schedule one slice per processor core, or per processor core orchestrating a GPU. The mesh in each slice is unstructured in the finite-element sense so that regions of interest, e.g., certain depths in the Earth where the earthquake occurred, are covered with more elements than other regions. The mesh as a whole is block-structured, i.e., each mesh slice is composed of an unstructured mesh, but all the mesh slices are topologically identical.

The resulting decomposition is topologically a regular grid and all the mesh slices and the cut planes, at which MPI neighborhood communication occurs, have the same number of elements and points. This implies that perfect load balancing is ensured between all the MPI processes.

3.2 Serial implementation

We use NVIDIA CUDA for our implementation, and refer to the CUDA documentation (NVIDIA Corporation (2009)) and conference tutorials (`http://gpgpu.org/developer`) for further information. In a CUDA program, the execution of a 'kernel' is manually partitioned into a so-called grid of thread blocks. Blocks within the grid cannot communicate with each other, while the threads in each block can synchronize via a small on-chip shared memory. The thread blocks are virtualized multiprocessors ('cores'), and are further partitioned automatically into 'warps' of 32, which execute in lockstep with a shared instruction pointer, i.e., in a single instruction multiple thread (superset of SIMD) fashion. If threads within a warp diverge in their execution path, the branches are serialized. Special care must be taken that the threads within a (half-) warp access contiguous regions in off-chip DRAM, to maximize the effective memory bandwidth because the hardware coalesces memory accesses into one transaction per half-warp under certain conditions.

The implementation of the serial case follows the algorithm outlined in Sect. 2.1 by mapping each of the following three steps into separate CUDA kernels. The first step updates the global displacement vector based on the previous time step, the second step performs the finite-element assembly, and the last step computes the global acceleration vector. The first and last steps are trivially parallel as they only affect the uniquely numbered data, and are mapped to CUDA in a straightforward fashion to automatically maximize multiprocessor occupancy and coalesce memory accesses into more efficient block transactions. In the following, we concentrate on the second step, as benchmarking reveals that it consumes more than 85 % of the time per time step (Komatitsch et al (2009)). We identify an element with a block of 128 threads (4 warps), and use one thread per cubature point. 125 out of 128 threads thus do useful work, and we avoid conditionals by zero padding. We first copy the global displacement vector corresponding to each element into shared memory using the global-to-local mapping. The derivative matrix of the Lagrange polynomials is stored in the so-called constant memory, and the kernel then multiplies it with the local coefficients of the displacement at the GLL points. Constant memory is cached, and it is as fast as registers if all threads access the same item simultaneously. The third stage performs numerical integration with the discrete Jacobian to obtain the local gradient of the displacement vector. In the final step, the elementwise contributions need to be assembled at each global point. Each such point receives contributions from a varying number of elements due to the non-structured mesh, which calls for an atomic summation, i.e., an order-independent sequential accumulation. We decouple these dependencies, which do not parallelize in a straightforward manner, by using a coloring scheme, resulting in one kernel call per color. Accordingly, we pre-compute maximally independent sets of mesh elements during the meshing step (see Sect. 3.1).

To maximize efficiency, we apply a number of CUDA-specific optimizations: Data is arranged so that accesses to local data stored in global memory (off-chip DRAM) can be coalesced into large memory transactions, and bank conflicts in shared memory are avoided. Accesses to arrays corresponding to global data cannot be fully coalesced due to the indirect addressing implied by the unstructured mesh, and we route these accesses through the texture cache to improve performance, although the improvement is rather small, as expected. A trade-off is required between the requirements of each block (due to the solution scheme) and the available resources, as shared memory and the register file in each multiprocessor are shared by resident blocks. We carefully tune the implementation so that at any given time, two blocks (8 warps) are concurrently active. This enables the hardware scheduler to switch threads when stalled at off-chip memory accesses, and results in better throughput. The final kernel is a result of repeated optimization and resource balancing every time new features have been added, because splitting the computation into two kernels and paging out all data to off-chip memory in-between is very expensive and should be avoided if at all possible. We refer to a previous publication for the technical details of the implementation omitted here due to page constraints (Komatitsch et al (2009)).

3.3 Parallel implementation

There are several challenges to address in mapping this computation to a GPU cluster. The elements that compose the mesh slices 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 these ‘outer’ elements, and an analogous list of the ‘inner’ elements. We compute the outer elements first, as it is done classically. Once these computations have been completed, we copy the associated data to the respective MPI buffers and issue a non-blocking MPI call, which initiates the communication and returns immediately. While the messages are traveling across the interconnect, 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 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 for classical CPU clusters, due to the speedup obtained by the GPUs.

The PCIe bus between the CPUs and the GPUs exhibits bandwidth and latency similar to an Infiniband interconnect. To alleviate this bottleneck, we insert two additional kernels before and after the loop over the element colors in the assembly process. The first one packs the contributions of the outer elements in the current slice into an auxiliary buffer. This buffer is transferred to the host, as PCIe transfers are much faster when performed in one large batch rather than in many small batches. The CPU unpacks the data, distributes it to the MPI buffers associated with the four neighboring slices and issues the non-blocking MPI call. A second auxiliary kernel performs the other way round, i.e., the unpacking of a PCIe transfer and the indirect writes to device memory. In our experiments, we found that this approach is faster overall than performing several PCIe transfers, despite the implied indirect reads and writes in device memory. CUDA allows for two alternative implementations to achieve overlap of PCIe communication and device computation. A feature called ‘streams’ is used in a way similar to our approach, where we decouple the outer from the inner elements at the MPI level (e.g., by overlapping computation of the elements sharing data in one direction with transfer of data for the other directions). This feature can be combined with ‘zero copy’, which maps a buffer on the CPU into device memory space. As the mesh is unstructured, the additional bookkeeping overhead is sufficient to nullify any performance improvements and thus we do not use these features.

4 Results

4.1 Test configurations

The machine we use is a cluster of 48 Teslas S1070 at CCRT/CEA/GENCI in Paris, 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 Nehalem processors operating at 2.93 GHz. Each node has 24 GB of RAM and runs Linux kernel 2.6.18. The network is Infiniband.

For the scalability tests, we use slices of 446,080 spectral elements each, out of which 122,068 are ‘outer’ elements, i.e., elements in contact with MPI cut planes by at least one mesh point, and 324,012 elements are ‘inner’ elements. The ratio between outer and inner elements is thus approximately 27.5% to 72.5 %. Each slice contains approximately 29.6 million unique grid points, i.e., 88.8 million degrees of freedom, corresponding to 3.6 GB (out of 4 GB) memory footprint per GPU. The largest possible problem size, using all 192 GPUs in the cluster, is thus 17 billion unknowns. All our measurements correspond to the duration (i.e., elapsed time) of 1,000 time steps, keeping in mind that each time step consists of the exact same numerical operations (see Sect. 2.1). To get accurate measurements, not subject to outside interference, the nodes that participate in a particular run are not shared with other users. Each run is executed three times to ensure that the timings are reliable, and to determine whether there are any fluctuations. The CPU reference code is heavily optimized (Komatitsch et al (2003, 2008)) using the ParaVer performance analysis tool, in particular to minimize cache misses.

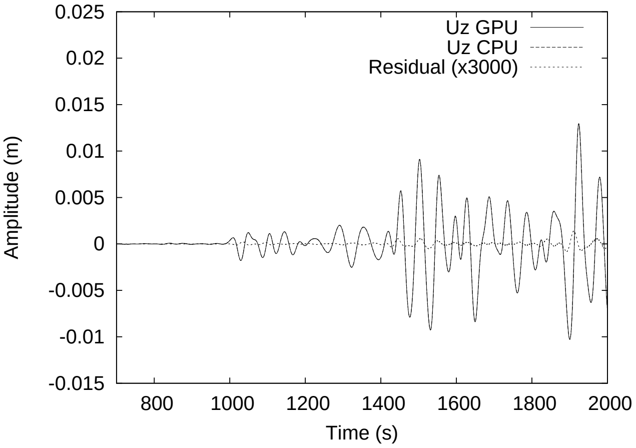

Fig. 1 We compare the result of our single-precision GPU + MPI code (solid line) and our reference existing single-precision CPU + MPI code (dashed line) for the time variation of the vertical component of the displacement vector at a given point in the mesh. The two curves are almost perfectly superimposed and the absolute difference amplified by a factor of 3,000 (dotted line) is very small.

4.2 Numerical validation

Spectral-element codes for linear seismic wave propagation modeling are always sufficiently accurate in single precision, as demonstrated, e.g., in Chaljub et al (2007), Tromp et al (2008) and Komatitsch et al (2009). It is therefore unnecessary to resort to double precision calculations to solve this problem, which is an advantage on current GPUs because single precision calculations are significantly faster, although the situation of double precision operations will improve, in particular on the FERMI architecture. In the remainder of this article, we thus use single precision on both the CPUs and the GPUs. Before we proceed with performance analysis of the GPU implementation, let us present a validation test that we performed for the GPU-accelerated code on 64 GPUs. Figure 1 shows a comparison of our single-precision GPU + MPI code and our reference existing single-precision CPU + MPI code for the time variation of the vertical component of the displacement vector at a given point in the mesh. Calculation of the absolute difference between both curves shows that the differences are negligible. This is the expected behavior because single precision is sufficient on CPUs and the different order of computation on the GPU does not lead to different results as the solution method is stable.

4.3 Weak scalability

Fig. 2 shows the average elapsed time per time step of the SEM algorithm for simulations on 4 to 192 GPUs (i.e., the whole machine), in steps of four GPUs. Weak scaling is close to perfect; the small fluctuations we observe are on

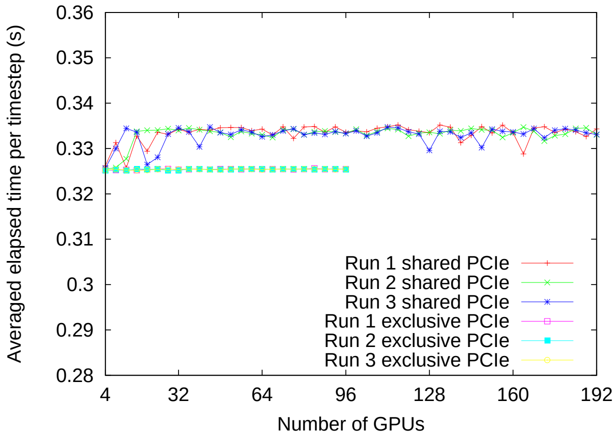

Fig. 2 Weak scalability using GPUs, with and without bus sharing.

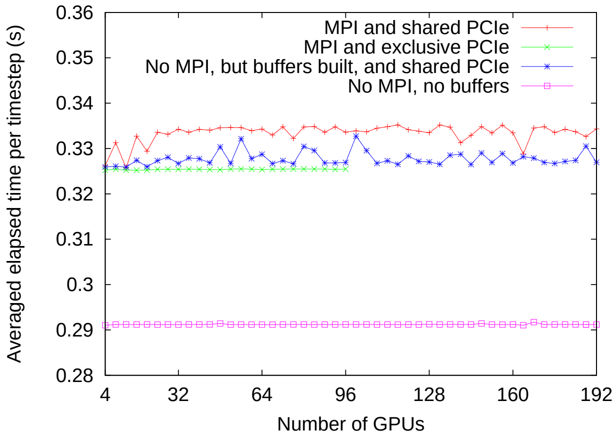

Fig. 3 Performance breakdown, assessment of MPI and PCIe overlap.

the order of 2–3 %. We repeat this experiment using only one GPU per node, and consequently, we can only go up to 96 GPUs when keeping the load per GPU fixed. The fluctuations are now entirely removed, which shows that all fluctuations are caused by the shared PCIe bus in each half-Tesla S1070. Furthermore, the entire run is on average only 3 % faster when the PCIe buses are not shared, i.e., the PCIe sharing implied by the design of the S1070 Tesla is not a dominant bottleneck. This demonstrates convincingly that the overlap of non-blocking MPI communication, PCIe transfers and computation on the devices is excellent.

We now examine in more detail the extent to which communications and computations overlap each other. Fig. 3 compares four sets of measurements (averaged over three runs in each case): the two curves of the last experiment, a calculation in which we completely turn off both the MPI communications and the creation and processing of the MPI buffers (magenta), and a calculation in which we create and process the MPI buffers but replace the MPI send/receives with zeroing of the buffers (blue). The last two configurations are used for this analysis only; they give incorrect results, but execute the same number of computations per time step. A comparison between the full calculation for the real problem with sharing of the PCIe bus (red tics) and the 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 (blue stars), provides a good estimate of the time spent waiting for communications (an upper bound of 3% of the time), i.e., this experiment is a good illustration of how effectively communications and calculations overlap. We also observe that the full calculation for the real problem without sharing of the PCIe bus (green crosses) constitutes a lower bound for the runtime, and several runs of the experiment depicted with blue stars fortuitously reach this bound. We can therefore conclude that all communications are perfectly overlapped with computations, and observed differences are due to PCIe bus sharing. A comparison with the modified calculation in which the MPI buffers are not built and MPI is completely turned off (magenta squares) gives an estimate of the total cost of running the problem on a cluster rather than a single core, i.e., 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 / 0.291 = 1.117 = 11.7%. We emphasize that this cost does not affect the efficiency (speedup) of the code, as it is not serialized. We did not try to estimate the cost of running the code without overlap between communication and computation since this would imply using blocked MPI, which creates a bottleneck when there is in excess of 2,000 cores, a situation anticipated on future large GPU clusters.

4.4 Speedup

To measure the speedup, we repeat the weak scaling experiment with two different CPU configurations. In the first one, we assign each slice of 3.6 GB to a CPU core, and to balance resource sharing with idle resources, we schedule two slices to each CPU. In the second one, we cut each slice in half and assign four of these smaller slices to the four cores in each CPU. We use process pinning to make sure that each MPI process uses its desired core exclusively. These experiments thus only require half the amount of cluster nodes, because more memory is available to each CPU than in each GPU board. Fig. 4 depicts the weak scaling measurements we obtain. The fluctuations are larger than in the CPU case (because the elapsed time is longer), but the relative amount of noise in the measurements is the same as in the pure GPU run. We currently cannot explain the repeatable peak in one of the configurations. The configuration that uses four cores per node to compute four half-sized slices is 1.6 times faster

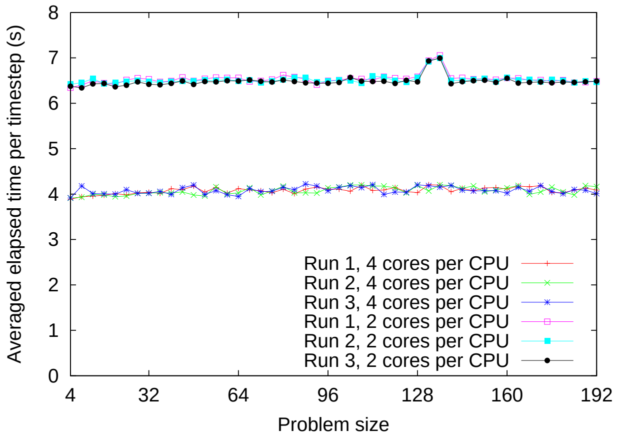

Fig. 4 Weak scalability using only CPU cores. Note that the two configurations compute the same problem, the problem size per CPU is the same (and the same as per GPU in Fig. 2).

than using only two cores for the full slices. We do not observe the ideal factor of two due to resource sharing.

When combining the measurements in Fig. 2 and Fig. 4, we can derive an average speedup of one GPU in a Tesla S1070 over four cores in a Nehalem CPU by a factor of 12.9, and of 20.6 over using only two CPU cores. Both factors are equally meaningful, because we need to halve the size of each slice for the eight-core run, which changes the ratio between inner and outer elements, and the communication pattern, size and amount of the MPI messages. In these types of experiments in geophysics, one is usually interested in running the largest problem size possible. We have therefore based our experiments as to fill up the device memory to 90 %.

5 Conclusions and future work

We have demonstrated excellent weak scalability of a high-order finite-element code to simulate seismic wave propagation on a cluster of 192 GPUs, and obtained speedup factors of more than an order of magnitude against a highly tuned CPU reference code, which performs the same number of arithmetic operations in the same precision, and computes an equally accurate result. To achieve full overlap of computation and communication (via MPI and between the host and the device via PCIe), non-blocking MPI in combination with a sufficiently large amount of device memory is necessary. The amount of communication that is not currently overlapped in our implementation is slightly more than 10 %, which is negligible in terms of weak scaling given the order-of-magnitude overall speedup that we obtain.

We deliberately did not evaluate hybrid CPU-GPU configurations, i.e., scheduling small batches of work to six CPU cores and using the remaining two to drive the GPUs which perform the majority of the calculations. Such a configuration would be imbalanced on this cluster because we cannot easily vary the size of the mesh slices, and more importantly because the speedup of one GPU over one CPU core is much higher than that obtained by the remaining CPU cores, even assuming perfect strong scaling. A hybrid parallelization that uses pThreads within each node and MPI only in between nodes would require a lot of coding effort, but in our opinion would yield only small returns in terms of overall efficiency.

In future work, we plan to re-implement our approach using OpenCL to expand the range of hardware on which the accelerated code will run, in particular to include AMD. We will also perform these experiments on the Lincoln machine in the Teragrid, which has CPUs from an older technology generation and a total of 384 of the same GPUs, which are connected via shared PCIe x8 (i.e., half the lanes compared to the machine used in these tests). We expect identical scalability results, but it would be interesting to evaluate how the different factors (PCIe lanes, PCIe sharing, NUMA effects etc.) influence performance and speedup. The new features of FERMI (NVIDIA's upcoming new GPU generation) are also worth further investigation: This chip almost quadruples the amount of shared memory per multiprocessor, and we expect the tuning effort when adding new features to our kernels to be significantly reduced. This is particularly important for more challenging and realistic physical simulations that would require anisotropic or viscoelastic geological media. Our current multi-GPU code is limited to the – still relevant in many problems of interest – case of isotropic elastic rheology (while our classical multi-CPU code has full support for anisotropy and viscoelasticity). Finally, we would like to assess the energy efficiency of the accelerated solver, i.e., measuring electrical energy consumption; we are discussing this possibility with BULL engineers.

Acknowledgements The authors would like to thank Jean Roman, Jean-François Méhaut, Christophe Merlet, Matthieu Ospici, Xavier Vigouroux and Roland Martin for fruitful discussion and CCRT/CEA and GENCI, France, for providing support and access to their 'Titane' BULL Novascale R422 GPU cluster.

References

Abdelkhalek R, Calandra H, Coulaud O, Roman J, Latu G (2009) Fast seismic modeling and reverse time migration on a GPU cluster. In: Smari WW, McIntire JP (eds) High Performance Computing & Simulation 2009, pp 36–44, http://hal.inria.fr/docs/00/40/39/33/PDF/hpcs.pdf

Carrington L, Komatitsch D, Laurenzano M, Tikir M, Michéa D, Le Goff N, Snavely A, Tromp J (2008) High-frequency simulations of global seismic wave propagation using SPECFEM3D_GLOBE on 62k processors. In:

SC ’08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, article #60, Gordon Bell Prize finalist article

Chaljub E, Komatitsch D, Vilotte JP, Capdeville Y, Valette B, Festa G (2007) Spectral element analysis in seismology. In: Wu RS, Maupin V (eds) Advances in wave propagation in heterogeneous media, Advances in Geophysics, vol 48, Elsevier - Academic Press, pp 365–419

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

De Basabe JD, Sen MK (2007) Grid dispersion and stability criteria of some common finite-element methods for acoustic and elastic wave equations. Geophysics 72(6):T81–T95, DOI 10.1190/1.2785046

Dziewoński AM, Anderson DL (1981) Preliminary reference Earth model. Physics of the Earth and Planetary Interiors 25(4):297–356, DOI 10.1016/0031-9201(81) 90046-7

Fan Z, Qiu F, Kaufman AE, Yoakum-Stover S (2004) GPU cluster for high performance computing. In: SC ’04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing, p 47, DOI 10.1109/SC.2004.26

Fatahalian K, Houston M (2008) A closer look at GPUs. Communications of the ACM 51(10):50–57, DOI 10.1145/1400181.1400197

Garland M, Grand SL, Nickolls J, Anderson JA, Hardwick J, Morton S, Phillips EH, Zhang Y, Volkov V (2008) Parallel computing experiences with CUDA. IEEE Micro 28(4):13–27, DOI 10.1109/MM.2008.57

Göddeke D, Strzodka R, Mohd-Yusof J, McCormick P, Buijssen SHM, Grajewski M, Turek S (2007) Exploring weak scalability for FEM calculations on a GPU-enhanced cluster. Parallel Computing 33(10-11):685–699, DOI 10.1016/j.parco.2007.09.002

Hughes TJR (1987) The finite element method, linear static and dynamic finite element analysis. Prentice-Hall International

Kindratenko VV, Enos JJ, Shi G, Showerman MT, Arnold GW, Stone JE, Phillips JC, Hwu W (2009) GPU clusters for high-performance computing. In: Proceedings on the IEEE Cluster’2009 Workshop on Parallel Programming on Accelerator Clusters (PPAC’09), pp 1–8

Komatitsch D, Coutel F, Mora P (1996) Tensorial formulation of the wave equation for modelling curved interfaces. Geophys J Int 127(1):156–168, DOI 10.1111/j.1365-246X.1996.tb01541.x

Komatitsch D, Tsuboi S, Ji C, Tromp J (2003) A 14.6 billion degrees of freedom, 5 teraflops, 2.5 terabyte earthquake simulation on the Earth Simulator. In: SC ’03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing, pp 4–11, DOI 10.1109/SC.2003.10023, Gordon Bell Prize winner article

Komatitsch D, Labarta J, Michéa D (2008) A simulation of seismic wave propagation at high resolution in the inner core of the Earth on 2166 processors of MareNostrum. In: High Performance Computing for Computational Science – VECPAR 2008, Springer, Lecture Notes in Computer Science, vol 5336, pp 364–377, DOI 10.1007/978-3-540-92859-1_33

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

Liu Q, Polet J, Komatitsch D, Tromp J (2004) Spectral-element moment tensor inversions for earthquakes in Southern California. Bulletin of the Seismological Society of America 94(5):1748–1761, DOI 10.1785/012004038

Michéa D, Komatitsch D (2010) Accelerating a 3D finite-difference wave propagation code using graphics cards. Geophysics Journal International In press.

Micikevicius P (2009) 3D finite-difference computation on GPUs using CUDA. In: GPGPU-2: Proceedings of the 2nd Workshop on General Purpose Processing on Graphics Processing Units, pp 79–84, DOI 10.1145/1513895.1513905

Nissen-Meyer T, Fournier A, Dahlen FA (2008) A 2-D spectral-element method for computing spherical-earth seismograms – II. Waves in solid-fluid media. Geophysics Journal International 174(3):873–888, DOI 10.1111/j.1365-246X.2008.03813.x

NVIDIA Corporation (2009) NVIDIA CUDA Programming Guide version 2.3. Santa Clara, California, USA, URL `http://www.nvidia.com/cuda`

Owens JD, Houston M, Luebke DP, Green S, Stone JE, Phillips JC (2008) GPU computing. Proceedings of the IEEE 96(5):879–899, DOI 10.1109/JPROC.2008.917757

Phillips EH, Zhang Y, Davis RL, Owens JD (2009) Rapid aerodynamic performance prediction on a cluster of graphics processing units. In: Proceedings of the 47th AIAA Aerospace Sciences Meeting, aIAA 2009-565

Phillips JC, Stone JE, Schulten K (2008) Adapting a message-driven parallel application to GPU-accelerated clusters. In: SC ’08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, DOI 10.1145/1413370.1413379, article No. 8

Seriani G, Priolo E (1994) A spectral element method for acoustic wave simulation in heterogeneous media. Finite Elements in Analysis and Design 16(3–4):337–348, DOI 10.1016/0168-874X(94)90076-0

Tromp J, Komatitsch D, Liu Q (2008) Spectral-element and adjoint methods in seismology. Communications in Computational Physics 3(1):1–32

van Wijk K, Komatitsch D, Scales JA, Tromp J (2004) Analysis of strong scattering at the micro-scale. Journal of the Acoustical Society of America 115(3):1006–1011, DOI 10.1121/1.1647480