Geosci. Model Dev. Discuss., 6, 3743–3786, 2013
www.geosci-model-dev-discuss.net/6/3743/2013/
doi:10.5194/gmdd-6-3743-2013

Geoscientific Model Development Discussions
Open Access

CUDA-C implementation of the ADER-DG method for linear hyperbolic PDEs

C. E. Castro[1,*], **J. Behrens**[2], **and C. Pelties**[3]

[1]Instituto de Alta Investigación, Universidad de Tarapacá, Casilla 7D Arica, Chile
[2]KlimaCampus, University of Hamburg, Grindelberg 5, 20144 Hamburg, Germany
[3]Department of Earth and Environmental Sciences, Geophysics Section, Ludwig-Maximilians-Universität, Munich, Germany
[*]formerly at: KlimaCampus, University of Hamburg, Hamburg, Germany

Received: 27 May 2013 – Accepted: 31 May 2013 – Published: 13 July 2013

Correspondence to: C. E. Castro (ccastro@uta.cl)

Published by Copernicus Publications on behalf of the European Geosciences Union.

Abstract

We implement the ADER-DG numerical method using the CUDA-C language to run the code in a Graphic Processing Unit (GPU). We focus on solving linear hyperbolic partial differential equations where the method can be expressed as a combination of precomputed matrix multiplications becoming a good candidate to be used on the GPU hardware. Moreover, the method is arbitrarily high-order involving intensive work on local data, a property that is also beneficial for the target hardware. We compare our GPU implementation against CPU versions of the same method observing similar convergence properties up to a threshold where the error remains fixed. This behaviour is in agreement with the CPU version but the threshold is larger that in the CPU case. We also observe a big difference when considering single and double precision where in the first case the threshold error is significantly larger. Finally, we did observe a speed up factor in computational time but this is relative to the specific test or benchmark problem.

1 Introduction

Many scientific research directions rely heavily on simulation-based knowledge gain, because experiments are either too costly or not possible. In those areas, where mechanical or fluid-dynamical processes play a role, these simulations often consist of numerical solutions of partial differential equations (PDEs). In particular, the time required to obtain the solution is of large importance in the quest for ever more reliable and more accurate research results. While fast and accurate computations can be achieved by advancement of algorithmic approaches, new hardware has also helped to solve ever larger and more complex problems. Here we concentrate on the second approach and employ specialised hardware for reducing computational time. In particular a General Purpose Graphics Processing Unit (GPGPU) (Owens et al., 2007) demands for carefully optimised algorithms but offer large gains in computational performance.

GMDD
6, 3743–3786, 2013

The NVIDIA Tesla C2070 card for example has a peak double precision floating point performance of 515 Gflops.

Graphics Processing Units (GPUs) were originally developed to assist the Central Processing Unit (CPU) of workstation type computers in processing all the data related to graphics. GPUs are based on the Single Instruction, Multiple Data (SIMD) programming paradigm where one instruction is used to compute a large set of similarly structured but distinct data, creating an important source of parallelism and therefore reducing computational time. In recent years, the developers of GPUs realised that scientific computation could benefit from this hardware architecture to obtain faster numerical solutions at low cost and started producing the GPGPUs together with a more user-friendly software interface.

The availability of standardised software and hardware triggered a number of new GPU implementations of numerical methods solving Navier-Stokes (Asouti et al., 2011), linear elasticity (Komatitsch et al., 2010; Rietmann et al., 2012; Mu et al., 2013), or shallow water (Brodtkorb et al., 2012; de la Asunción et al., 2012) equations among others. The leading facilities in the current TOP500 list of supercomputers comprise GPGPU architectures (Meuer et al., 2012), and it is foreseeable that these types of machines will prevail in the coming years.

In this manuscript we present a numerical implementation of the ADER-DG (Käser and Dumbser, 2006; Castro et al., 2010) numerical method on the CUDA (formerly Compute Unified Device Architecture) parallel programming model from NVIDIA, in particular in C with CUDA extensions. Other language options are e.g. OpenCL, CUDA-Fortran, CUDA-Python. We will focus on C-CUDA because it is open and commonly used. We will use the current generation Fermi-based Tesla card C2070, which supports double precision IEEE 754 standard floating point arithmetic, to solve a hyperbolic linear partial differential equation with variable coefficients of the form

$$\frac{\partial u(\boldsymbol{x},t)}{\partial t} + \frac{\partial f(\boldsymbol{x},t)}{\partial x} + \frac{\partial g(\boldsymbol{x},t)}{\partial y} = 0\,, \tag{1}$$

GMDD
6, 3743–3786, 2013

with $f(\boldsymbol{x},t) = a(\boldsymbol{x})u(\boldsymbol{x},t)$ and $g(\boldsymbol{x},t) = b(\boldsymbol{x})u(\boldsymbol{x},t)$ the flux functions in x and y directions, respectively. In vectorial notation

$$\frac{\partial \mathbf{U}(\boldsymbol{x},t)}{\partial t} + \frac{\partial \mathbf{F}(\boldsymbol{x},t)}{\partial x} + \frac{\partial \mathbf{G}(\boldsymbol{x},t)}{\partial y} = \mathbf{0}, \tag{2}$$

with $\mathbf{F}(\boldsymbol{x},t) = \mathbf{A}(\boldsymbol{x})\mathbf{U}(\boldsymbol{x},t)$ and $\mathbf{G}(\boldsymbol{x},t) = \mathbf{B}(\boldsymbol{x})\mathbf{U}(\boldsymbol{x},t)$ the flux vectors in x and y direction, respectively. We use an arbitrarily high-order in space and time Discontinuous Galerkin numerical method. We said arbitrarily in the sense that the order of the method is an input parameter in the implementation and that there is no theoretical limit for it. In practice, the maximum achievable order depends on machine accuracy and is strongly limited by memory and CPU time resources.

PDE systems of the form (2) represent (between others) the advection of a vector $\mathbf{U}(\boldsymbol{x},t)$ as a consequence of a velocity field given by the entries of matrices $\mathbf{A}(\boldsymbol{x})$ and $\mathbf{B}(\boldsymbol{x})$. This velocity field is space dependent and has not a divergence free constraint. Equation (2) is used for tracer advection or linear elasticity in its velocity-stress formulation.

The mentioned ADER-DG numerical method has several desired properties: it is high-order accurate in space and time and therefore numerical errors are minimized; it can be implemented on unstructured meshes which allows us to consider complex geometries; it makes use of a reference spatial coordinate system, where basis functions are defined, and therefore many integrals can be pre-computed; and it uses a small stencil to communicate to neighbour elements requiring only direct neighbours for the flux computation independent of the applied order. In summary, the discrete version of the numerical method can be formulated as a set of matrix-matrix multiplication steps and is therefore a good candidate to be implemented on the GPU hardware.

It should be noted that the method presented is capable of utilizing non-uniform and unstructured meshes. While in Sect. 4 we present results for triangular meshes, the method works for triangular and quadrilateral elements. This is in contrast to many CUDA implementations, that make use of the simplicity of structured rectangular and uniform meshes and corresponding computational stencils.

GMDD
6, 3743–3786, 2013

In this manuscript we try to understand if the GPU implementation of the ADER-DG numerical method retains the well-known high-order properties documented in Käser and Dumbser (2006); Castro et al. (2010); Pelties et al. (2010); Hermann et al. (2011); Pelties et al. (2012); SeisSol (2013) between others.

The manuscript is organized as follows: in Sect. 2 we describe the numerical method, in Sect. 3 we briefly describe the hardware and explain the data structure to best fit in the GPU architecture, in Sect. 4 we show numerical results where we compare and assess the new CUDA implementation and finally in Sect. 5 we discuss the results and show the conclusions.

2 Numerical method

In this section we describe the derivation of the numerical method which follows on previous developments presented in the literature by Dumbser (2005); Käser and Dumbser (2006); Castro et al. (2010); Pelties et al. (2010); Hermann et al. (2011); Pelties et al. (2012) among others. The numerical method is constructed based on a spatial discretization of the physical domain $\Omega \in \mathbb{R}^2$ considering a conforming discretization where we use super-index i to identify element (triangle or quadrangle) $E^i \in \Omega$. The p-th component of the unknown vector $\mathbf{U}(\boldsymbol{x},t)$ is approximated using a linear combination of time-dependent degrees of freedom (dof) $\hat{u}_{pl}(t)$ and space-dependent basis functions $\phi_l(\xi,\eta)$ which are defined in a reference coordinate system (ξ,η). See Appendix A for details on the basis functions and mapping from physical to reference space coordinate systems. For $\boldsymbol{x} \in E^i$ we approximate the p-th component of $\mathbf{U}(\boldsymbol{x},t)$ as

$$u_p(\boldsymbol{x},t) \approx \sum_{l=1}^{N_l} \hat{u}^i_{pl}(t)\,\phi_l(\xi,\eta), \tag{3}$$

GMDD
6, 3743–3786, 2013

with N_l the number of basis functions used to approximate the continuous function $u_p(\boldsymbol{x},t)$ and defined by

$$N_l = \frac{\mathcal{O}(\mathcal{O}+1)}{2} \tag{4}$$

where $\mathcal{O}$ is the approximation order of the numerical method. In what follows we will drop the space and time dependences to simplify notation.

Taking the p-th component of Eq. (2), multiplying by the base function ϕ_k which is in the same space as ϕ_l and integrating in space over the i-th element we have

$$\int_{E^i} \phi_k \left[\partial_t u_p + \partial_x f_p + \partial_y g_p\right] \mathrm{d}V = 0\,, \tag{5}$$

or using the divergence operator

$$\int_{E^i} \phi_k \left[\partial_t u_p + \nabla \cdot h_p\right] \mathrm{d}V = 0\,. \tag{6}$$

Here h_p is the p-th component of $\mathbf{H} = [\mathbf{F}, \mathbf{G}]$. From the vector calculus identity $\nabla \cdot (\phi_k \mathbf{H}) = \phi_k (\nabla \cdot \mathbf{H}) + \mathbf{H} \cdot \nabla \phi_k$ and using the divergence theorem we obtain

$$\int_{E^i} \phi_k \partial_t u_p \,\mathrm{d}V - \int_{E^i} h_p \cdot (\nabla \phi_k)\,\mathrm{d}V + \int_{\partial E^i} \left(\phi_k h_p\right) \cdot \boldsymbol{n}\,\mathrm{d}S = 0\,. \tag{7}$$

Using Einstein notation we write $h_p = [f_p, g_p] = [A_{pq}, B_{pq}]\, u_q$. Expanding the derivatives on the second term in Eq. (7) and ordering them we have

$$\int_{E^i} \phi_k \partial_t u_p \,\mathrm{d}V - \int_{E^i} h^*_p \cdot \left(\nabla^{\xi} \phi_k\right) \mathrm{d}V + \int_{\partial E^i} \left(\phi_k h_p\right) \cdot \boldsymbol{n}\,\mathrm{d}S = 0\,, \tag{8}$$

GMDD
6, 3743–3786, 2013

with,

$$h_p^* = \left[\xi_x A_{pq} + \xi_y B_{pq}\,,\, \eta_x A_{pq} + \eta_y B_{pq}\right] u_q$$
$$= \left[A_{pq}^*\,,\, B_{pq}^*\right] u_q\,. \tag{9}$$

Here we used the chain rule to expand the derivatives and define $\nabla^{\xi} = [\partial_{\xi}, \partial_{\eta}]$. Note that we use the * upper index to identify the matrices written in the reference space coordinates. From Eq. (2) we see that $\mathbf{A}(\boldsymbol{x})$ and $\mathbf{B}(\boldsymbol{x})$ are spatially dependent matrices therefore we can approximate them in the same manner as Eq. (3) writing

$$A_{pq}^i(\boldsymbol{x}) \approx \sum_{m=1}^{N_m} \hat{A}_{pqm}^i \phi_m(\xi,\eta)\,,$$
$$B_{pq}^i(\boldsymbol{x}) \approx \sum_{m=1}^{N_m} \hat{B}_{pqm}^i \phi_m(\xi,\eta)\,, \tag{10}$$

with $N_m = \mathcal{O}_m(\mathcal{O}_m+1)/2$ where $\mathcal{O}_m \leq \mathcal{O}$ is the approximation order for the matrices. This means that we can change the approximation order used to represent the wind field for the advection equation or the material properties in the elastic wave equation. See Castro et al. (2010) for sub-cell resolution approximation.

Assuming that the Jacobian of the mapping $J = \partial(x,y)/\partial(\xi,\eta)$ is constant inside element E^i (see Appendix A for details on the mapping) we can easily compute h_p^* considering that

$$A_{pq}^{*i} \approx \hat{A}_{pqm}^{*i}\phi_m \text{ and } B_{pq}^{*i} \approx \hat{B}_{pqm}^{*i}\phi_m \tag{11}$$

are obtained from Eqs. (9) and (10). Now we substitute Eqs. (3) and (11) into Eq. (8) and express the volume integrals in the reference element E^{R} with $|J|$ due to the change

GMDD
6, 3743–3786, 2013

in the integration domain,

$$\partial_t \hat{u}^i_{pl} M_{kl} |J| - \hat{A}^{*i}_{pqm} K^{k\xi}_{klm} \hat{u}^i_{ql} |J| - \hat{B}^{*i}_{pqm} K^{k\eta}_{klm} \hat{u}^i_{ql} |J| + \int_{\partial E^i} (\phi_k h_p) \cdot \boldsymbol{n} \, \mathrm{d}S = 0 , \tag{12}$$

with pre-computed matrices

$$M_{kl} = \int_{E^R} \phi_k \phi_l \, \mathrm{d}V ,$$

$$K^{k\xi}_{klm} = \int_{E^R} \partial_\xi \phi_k \phi_l \phi_m \, \mathrm{d}V , \tag{13}$$

$$K^{k\eta}_{klm} = \int_{E^R} \partial_\eta \phi_k \phi_l \phi_m \, \mathrm{d}V .$$

The boundary integral in Eq. (12) is the flux contribution from neighbour elements and it is computed using a numerical flux function.

2.1 Numerical flux

The numerical flux for a linear PDE can be expressed as a linear function of the unknown vectors from the two adjacent elements $\mathbf{U}^i$ and $\mathbf{U}^{ij}$, with j counting the direct

GMDD
6, 3743–3786, 2013

neighbour elements as indicated in Fig. 1,

$$\begin{aligned}\mathbf{H}\cdot\boldsymbol{n} &= [\mathbf{F},\mathbf{G}]\cdot[n_x,n_y],\\ &= \left[\frac{(\mathbf{A}+|\mathbf{A}|)}{2}\mathbf{U}^i + \frac{(\mathbf{A}-|\mathbf{A}|)}{2}\mathbf{U}^{ij}\right] n_x\\ &+ \left[\frac{(\mathbf{B}+|\mathbf{B}|)}{2}\mathbf{U}^i + \frac{(\mathbf{B}-|\mathbf{B}|)}{2}\mathbf{U}^{ij}\right] n_y,\end{aligned} \tag{14}$$

where $|\mathbf{A}| = \mathbf{R}|\Lambda|\mathbf{R}^{-1}$ with $\mathbf{R}$ the right eigenvectors and $|\Lambda| = \mathrm{diag}(|\lambda_1|,|\lambda_2|,\ldots)$ a diagonal matrix with absolute values of the eigenvalues on the diagonal. If we evaluate $|\mathbf{A}| = \mathrm{diag}(\lambda_A^+)$, with λ_A^+ the maximum positive eigenvalue of matrix $\mathbf{A}$, we obtain the well known Rusanov flux (Rusanov, 1970). The same procedure is used for the matrix $\mathbf{B}$. Using the Rusanov flux the numerical flux function is expressed as

$$\begin{aligned}\mathbf{H}\cdot\boldsymbol{n} &= \tfrac{1}{2}\left(\mathbf{A}n_x + \mathbf{B}n_y + \mathbf{S}^+\right)\mathbf{U}^i\\ &+ \tfrac{1}{2}\left(\mathbf{A}n_x + \mathbf{B}n_y - \mathbf{S}^+\right)\mathbf{U}^{ij},\end{aligned} \tag{15}$$

with $\mathbf{S}^+ = \mathbf{I}|n_x\lambda_A^+ + n_y\lambda_B^+|$ and $\mathbf{I}$ the identity matrix. Writing Eq. (15) in tensor notation we have

$$\begin{aligned}h_p\cdot\boldsymbol{n} &= \tfrac{1}{2}\left(A_{pq}n_x + B_{pq}n_y + S^+_{pq}\right)u_q^i\\ &+ \tfrac{1}{2}\left(A_{pq}n_x + B_{pq}n_y - S^+_{pq}\right)u_q^{ij}.\end{aligned} \tag{16}$$

Before substituting the Jacobian approximations (10) into Eq. (16) we choose to evaluate λ_A^+ and λ_B^+ based on the mean value of the corresponding matrices. As a consequence, S^+_{pq} is not space dependent and only varies on the edge considered

GMDD
6, 3743–3786, 2013

in the numerical flux function. For implementation purposes we still use a polynomial expansion of it but knowing that all but the first degree of freedom $\hat{S}^{+}_{pqm}$ are zeros.

Now the space dependent flux function is written as follows

$$h_p \cdot \boldsymbol{n} = \frac{1}{2}\left(\hat{A}_{pqm}n_x + \hat{B}_{pqm}n_y + \hat{S}^{+}_{pqm}\right)\hat{u}^{i}_{ql}\phi_l\phi_m + \frac{1}{2}\left(\hat{A}_{pqm}n_x + \hat{B}_{pqm}n_y - \hat{S}^{+}_{pqm}\right)\hat{u}^{ij}_{ql}\phi_l\phi_m . \tag{17}$$

Finally the boundary integral in Eq. (12) responsible for the flux computation is discretized by

$$\int_{\partial E^i} (\phi_k h_p)\cdot\boldsymbol{n}\,\mathrm{d}S = \sum_{j=1}^{N_s}\left[\frac{1}{2}\left(\hat{A}_{pqm}n^j_x + \hat{B}_{pqm}n^j_y + \hat{S}^{+}_{pqm}\right)\hat{u}^{i}_{ql}\right]|S^j|\mathcal{F}^{j,0}_{klm} + \sum_{j=1}^{N_s}\left[\frac{1}{2}\left(\hat{A}_{pqm}n^j_x + \hat{B}_{pqm}n^j_y - \hat{S}^{+}_{pqm}\right)\hat{u}^{ij}_{ql}\right]|S^j|\mathcal{F}^{j,i}_{klm}, \tag{18}$$

where $|S^j|$ is the length of the j-th edge and N_s is the number of edges depending on triangular or quadrilateral elements. The boundary integrals $\mathcal{F}^{j,0}_{klm}$ and $\mathcal{F}^{j,i}_{klm}$ are precomputed in the reference space coordinates using the parameter $\chi_j \in [0,1]$ describing

GMDD
6, 3743–3786, 2013

the j-th edge,

$$\mathcal{F}^{j,0}_{klm} = \int_0^1 \phi_k(\chi_j)\phi_l(\chi_j)\phi_m(\chi_j)d\chi_j\,,$$
$$\mathcal{F}^{j,i}_{klm} = \int_0^1 \phi_k(\chi_j)\phi^{ij}_l(1-\chi_j)\phi_m(\chi_j)d\chi_j\,. \tag{19}$$

Note that $\mathcal{F}^{j,0}_{klm}$ is responsible for the flux contribution from the element E^i while $\mathcal{F}^{j,i}_{klm}$ considers the contribution of the neighbour element as the neighbour degrees of freedom are considered in the second integral by $\phi^{ij}_l(1-\chi_j)$. Now the semi-discrete scheme is written emphasizing the remaining time dependence

$$\partial_t \hat{u}^i_{pl}(t) M_{kl}|J| =$$
$$\left[\hat{A}^{*i}_{pqm}K^{k\xi}_{klm} + \hat{B}^{*i}_{pqm}K^{k\eta}_{klm}\right]|J|\hat{u}^i_{ql}(t)$$
$$-\sum_{j=1}^{N_s}\frac{1}{2}\left[\hat{A}_{pqm}n^j_x + \hat{B}_{pqm}n^j_y + \hat{S}^+_{pqm}\right]|S^j|\mathcal{F}^{j,0}_{klm}\hat{u}^i_{ql}(t) \tag{20}$$
$$-\sum_{j=1}^{N_s}\frac{1}{2}\left[\hat{A}_{pqm}n^j_x + \hat{B}_{pqm}n^j_y - \hat{S}^+_{pqm}\right]|S^j|\mathcal{F}^{j,i}_{klm}\hat{u}^{ij}_{ql}(t)\,.$$

GMDD
6, 3743–3786, 2013

Integrating Eq. (20) in time

$$\left[\hat{u}_{pl}^{i,n+1} - \hat{u}_{pl}^{i,n}\right] M_{kl}|J| =$$

$$\left[\hat{A}^{*i}_{pqm} K^{k\xi}_{klm} + \hat{B}^{*i}_{pqm} K^{k\eta}_{klm}\right] |J| \int_{t^n}^{t^{n+1}} \hat{u}^{i}_{ql}(\tau)\,\mathrm{d}\tau$$

$$- \sum_{j=1}^{N_s} \frac{1}{2} \left[\hat{A}_{pqm} n_x^j + \hat{B}_{pqm} n_y^j + \hat{S}^{+}_{pqm}\right] |S^j| \mathcal{F}^{j,0}_{klm} \int_{t^n}^{t^{n+1}} \hat{u}^{i}_{ql}(\tau)\,\mathrm{d}\tau \tag{21}$$

$$- \sum_{j=1}^{N_s} \frac{1}{2} \left[\hat{A}_{pqm} n_x^j + \hat{B}_{pqm} n_y^j - \hat{S}^{+}_{pqm}\right] |S^j| \mathcal{F}^{j,i}_{klm} \int_{t^n}^{t^{n+1}} \hat{u}^{ij}_{sl}(\tau)\,\mathrm{d}\tau\,,$$

we obtain the discrete scheme to update the numerical solution from time level $t = t^n$ to time level $t = t^{n+1} = t^n + \Delta t$. The time step Δt is defined from the stability condition $\Delta t = \mathrm{CFL}\, \Delta x / S_{\max}$.

The time integral $\int_{t^n}^{t^{n+1}} \hat{u}^{i}_{pl}(\tau)\,\mathrm{d}\tau$ is obtained from the Cauchy-Kowalewski procedure and explained in the following section.

2.2 Time integration

This time integration scheme is based on a local space-time expansion that is valid inside the element E^i for one time step Δt and is constructed as follows.

GMDD
6, 3743–3786, 2013

Writing Eq. (2) in the reference space coordinates, multiplying by the basis function ϕ_k and integrating over the reference element yields

$$\int_{E^R} \phi_k \left[\partial_t \hat{u}_p + A^*_{pq}\left(\partial_\xi u_q\right) + B^*_{pq}\left(\partial_\eta u_q\right) + \left(\partial_\xi A^*_{pq}\right) u_q + \left(\partial_\eta B^*_{pq}\right) u_q\right] \mathrm{d}V = 0\,. \tag{22}$$

Introducing the polynomial approximation defined in Eqs. (3) and (10) we find an expression for the first time derivative of the degrees of freedom

$$\partial_t \hat{u}^i_{pl}(t) M_{kl} = -\left[\hat{A}^{*i}_{pqm} K^{l\xi}_{klm} + \hat{B}^{*i}_{pqm} K^{l\eta}_{klm} + \hat{A}^{*i}_{pqm} K^{m\xi}_{klm} + \hat{B}^{*i}_{pqm} K^{m\eta}_{klm}\right] \hat{u}^i_{ql}(t)\,. \tag{23}$$

Applying recursively this algorithm we can estimate any order time-derivative using

$$\frac{\partial^r \hat{u}^i_{pl}(t)}{\partial t^r} M_{kl} = -\left[\hat{A}^{*i}_{pqm} K^{l\xi}_{klm} + \hat{B}^{*i}_{pqm} K^{l\eta}_{klm} + \hat{A}^{*i}_{pqm} K^{m\xi}_{klm} + \hat{B}^{*i}_{pqm} K^{m\eta}_{klm}\right] \frac{\partial^{r-1} \hat{u}^i_{ql}(t)}{\partial t^{r-1}}\,. \tag{24}$$

Now we can approximate the time evolution of the numerical solution for $\tau \in [0, \Delta t]$ using a Taylor expansion

$$\hat{u}^i_{pl}(t^n + \tau) = \sum_{r=0}^{\mathcal{O}-1} \frac{\partial^r \hat{u}^i_{pl}(t^n)}{\partial t^r} \frac{\tau^r}{r!} \tag{25}$$

GMDD
6, 3743–3786, 2013

where $\mathcal{O}$ is the order of the numerical method. The time integral of the degrees of freedom is easily computed from Eq. (25) to obtain

$$\int_0^{\Delta t} \hat{u}_{pl}^i(t^n+\tau)\,\mathrm{d}\tau = \sum_{r=0}^{\mathcal{O}-1} \frac{\partial^r \hat{u}_{pl}^i(t^n)}{\partial t^r} \frac{\Delta t^{r+1}}{(r+1)!}. \tag{26}$$

Using Eq. (26) we can update the numerical solution (21) in one single step from $t = t^n$ to $t = t^{n+1}$.

2.3 Sub model

In the previous sections we developed the numerical method to solve Eq. (2) that represents, for example, the advection of a tracer driven by a velocity field which can accelerate and therefore is spatially dependent and non divergence-free. In geophysical science, we found another very interesting partial differential equation that represents the propagation of seismic waves, written in the form of the linear elastic wave equation. In its velocity-stress formulation (Virieux, 1984, 1986) is a linear hyperbolic problem which can not be written in conservative form, instead is formulated in the quasi-conservative form as presented in Käser and Dumbser (2006); Castro et al. (2010),

$$\frac{\partial \mathbf{U}(\boldsymbol{x},t)}{\partial t} + \mathbf{A}(\boldsymbol{x})\frac{\partial \mathbf{U}(\boldsymbol{x},t)}{\partial x} + \mathbf{B}(\boldsymbol{x})\frac{\partial \mathbf{U}(\boldsymbol{x},t)}{\partial y} = \mathbf{0}. \tag{27}$$

Here, matrices $\mathbf{A}(\boldsymbol{x})$ and $\mathbf{B}(\boldsymbol{x})$ appear outside the respective spatial derivatives. In the limit case where $\mathbf{A}(\boldsymbol{x}) \equiv \mathbf{A}$ and $\mathbf{B}(\boldsymbol{x}) \equiv \mathbf{B}$ are constant, Eqs. (2) and (27) are equivalent. Otherwise, they represent a different problem and therefore have different solutions. One can move between both formulations making use of a source term. See LeVeque (2002) for more details on variable-coefficient linear equations. Here we briefly demonstrate the modifications required to accommodate for solving Eq. (27).

It is possible to write Eq. (27) in conservative form by adding the missing part of the conservative products and therefore introducing a new source term. The final equation

GMDD
6, 3743–3786, 2013

which is equivalent to Eq. (27) reads

$$\frac{\partial \mathbf{U}(\boldsymbol{x},t)}{\partial t} + \frac{\partial \mathbf{F}(\boldsymbol{x},t)}{\partial x} + \frac{\partial \mathbf{G}(\boldsymbol{x},t)}{\partial y} = \frac{\partial \mathbf{A}(\boldsymbol{x})}{\partial x}\mathbf{U}(\boldsymbol{x},t) + \frac{\partial \mathbf{B}(\boldsymbol{x})}{\partial y}\mathbf{U}(\boldsymbol{x},t), \tag{28}$$

with $\mathbf{F}(\boldsymbol{x},t) = \mathbf{A}(\boldsymbol{x})\mathbf{U}(\boldsymbol{x},t)$ and $\mathbf{G}(\boldsymbol{x},t) = \mathbf{B}(\boldsymbol{x})\mathbf{U}(\boldsymbol{x},t)$.

The introduction of the right hand side source term implies two modifications to our numerical method. First, the time integration scheme from Eq. (24) is replaced by

$$\frac{\partial^r \hat{u}^i_{pl}(t)}{\partial t^r} M_{kl} = -\left[\hat{A}^{*i}_{pqm} K^{l\xi}_{klm} + \hat{B}^{*i}_{pqm} K^{l\eta}_{klm}\right] \frac{\partial^{r-1} \hat{u}^i_{ql}(t)}{\partial t^{r-1}}. \tag{29}$$

Second, the update Eq. (21) needs to be modified incorporating the source contribution as follows

$$\begin{aligned}
&\left[\hat{u}^{i,n+1}_{pl} - \hat{u}^{i,n}_{pl}\right] M_{kl}|J| = \\
&\left[\hat{A}^{*i}_{pqm} K^{k\xi}_{klm} + \hat{B}^{*i}_{pqm} K^{k\eta}_{klm}\right] |J| \int_{t^n}^{t^{n+1}} \hat{u}^i_{ql}(\tau)\,\mathrm{d}\tau \\
&- \sum_{j=1}^{N_s} \frac{1}{2}\left[\hat{A}_{pqm} n^j_x + \hat{B}_{pqm} n^j_y + \hat{S}^+_{pqm}\right] |S^j| \mathcal{F}^{j,0}_{klm} \int_{t^n}^{t^{n+1}} \hat{u}^j_{ql}(\tau)\,\mathrm{d}\tau \\
&- \sum_{j=1}^{N_s} \frac{1}{2}\left[\hat{A}_{pqm} n^j_x + \hat{B}_{pqm} n^j_y - \hat{S}^+_{pqm}\right] |S^j| \mathcal{F}^{j,i}_{klm} \int_{t^n}^{t^{n+1}} \hat{u}^{jj}_{sl}(\tau)\,\mathrm{d}\tau \\
&+ \left[\hat{A}^{*i}_{pqm} K^{m\xi}_{klm} + \hat{B}^{*i}_{pqm} K^{m\eta}_{klm}\right] |J| \int_{t^n}^{t^{n+1}} \hat{u}^i_{ql}(\tau)\,\mathrm{d}\tau .
\end{aligned} \tag{30}$$

GMDD
6, 3743–3786, 2013

3 Numerical implementation

3.1 Hardware characteristics, grid, block and thread

In this implementation we use the NVIDIA Tesla C2070 graphic card which has 14 Streaming Multiprocessors (SM) each with 32 Streaming Processors (SP) also called CUDA cores represented in Fig. 2. While this is the physical distribution inside the GPU, a developer needs to design the code considering another abstraction model. The general outline of the code to be sent to the GPU is following the SIMD (Single Instruction, Multiple Data) programming model.

In a more detailed view the programming primitives are defined by grids, blocks and threads as shown in Fig. 3 where finally the CUDA-C functions, called kernels, run. Beside this we need to consider resources like global memory, shared memory and registers among others which are limited in size and bandwidth/latency, see NVidia (Consulted Nov 2012b) for more details.

Once an algorithm is written in a kernel, multiple versions of the same kernel are executed acting on different data (in accordance with the SIMD model). On runtime, each of these kernels is called a thread and together define the thread block. A thread block is assigned to a SM to be executed and inside this block the shared memory is available to all threads. The collection of all blocks is defined by the grid as shown in Fig. 3. In the same code line where a kernel is called, the CUDA extension symbols "<<<" and ">>>" are used to define the dimensions of the grid and the block as show in Fig. 5.

3.2 Data structure

In order to use the GPU architecture we need to design a data structure such that numerical algorithms run efficiently. In this manuscript we assume that the main time loop of our computation can be entirely coded and executed in the GPU memory without copying data between CPU and GPU memory in runtime. Moreover we mainly

GMDD
6, 3743–3786, 2013

consider locality of the data as our means of optimisation. Other more sophisticated approaches can deal with data alignment, cache miss minimization and many more, as explained by NVidia (Consulted Nov 2012a).

In Eq. (3) we introduced the polynomial expansion for the unknown vector $\mathbf{U}_p$, where p stands for the p-th component of the PDE, approximating the solution with a linear combination of basis function ϕ_l and degrees of freedom $\hat{u}_{pl}$, with l representing the corresponding basis function. The numerical method makes intense use of the degrees of freedom in the update scheme of one element as expressed in Eq. (21). In order to use the dofs inside of our GPU algorithm we will store them in one single array of dimension 2. We call this array `dof` and the size is `[nDof_l, nComp x nElem]`. The first index correspond to the number of degrees of freedom used to expand the unknown vector, while the second index is obtained from the number of components of the PDE times the number of elements in the mesh.

In this form, when this array is stored inside GPU global memory, the degrees of freedom associated to element E^i are stored in the position `nDof_l*nComp*(i-1) ... nDof_l*nComp*(i)` and therefore adjacent in the memory. We follow the same approach for the degrees of freedom of the Jacobian matrices in Eq. (11).

3.3 GPU algorithm

The CUDA algorithm used to implement the ADER-DG numerical method is organised to be contained in a shared library so we can recycle most of our existing Fortran code. The new CUDA code is structured in two parts. The first one is the driver which is called by the Fortran code after all preprocess is performed. The driver is responsible for preparing the data structure such that to optimise the GPU memory and to upload all required information into the GPU memory. Schematically this algorithm is represented in Fig. 5. We use two kernels to implement the numerical method. The first one is called `TimeIntegrateDof` and is responsible for the time integration step (26) and

GMDD
6, 3743–3786, 2013

the volume integral. The second kernel is called `FluxComputation`, which depends on the previous one, and is responsible for the numerical flux computation Eq. (18).

The kernel definition is designed such that we map each element E^i to a thread block in a 2-D array. Each block then is defined to map the degrees of freedom $\hat{u}^i_{lp}$ inside each element as graphically represented in Fig. 3. The parameters `dimGrid` and `dimBlock` (see Fig. 5) are defined during run time in order to optimize the kernel execution. Moreover we use two constants that are hard coded in a C header file such that we can define the size of the shared memory used. These two parameters are the number of components of the PDE and the number of dof given by Eq. (4).

```
#define MAX_nCOMP 5 #define MAX_nDOF 21
```

In this case we define them for the case of the seismic wave equation and sixth order method in two spatial dimensions. To construct a more general software one can consider to use templates but this is out of the scope of the current work.

Using this configuration we use the following C-CUDA intrinsic functions `gridDim`, `blockIdx` and `threadIdx` to identify the element (iElem), component (ip) and degrees of freedom (il) inside each kernel.

```
iElem = blockIdx.y*gridDim.x + blockIdx.x; ip = threadIdx.x; il =
threadIdx.y;
```

The reader can find the implementation of these kernels as a complementary material to this manuscript.

3.4 Memory consumption

Because our scope is to run this implementation completely inside the GPU, we need to be sure that the available memory is capable to store all the required information. To this end we compute the theoretical memory consumption for a 5 component PDE (linear elasticity in 2-D for example) considering double precision and order of approximation from first to seventh, that is 1 to 28 degrees of freedom. In Fig. 6 we plot the

GMDD
6, 3743–3786, 2013

results where horizontal axis is number of elements and vertical axis is memory. The maximum GPU memory of the Tesla C2070 cards is 6 Gb therefore we could easily fit up to 3×10^9 elements with fifth (p4) order of accuracy.

4 Numerical results

In this section we will discuss the numerical results of our GPU implementation of the ADER-DG scheme. For all results obtained from the implemented algorithm we used a fixed time step obtained from the CFL condition

$$\Delta T = \min \left\{ \frac{\mathrm{CFL}\, \Delta x^i}{(2\mathcal{O}-1)S^i} \right\} \quad \text{for all } E^i \in \Omega, \tag{31}$$

with S^i the maximum wave speed inside element E^i, Δx^i measured as the minimum distance from the barycentre to the edge of the element and CFL = 0.9.

To assess the proposed implementation we perform several test problems based on measuring the order of convergence. The empirically determined order of convergence (with respect to the mesh size) allows us to verify the expected accuracy of the numerical method and to identify code implementation errors. A convergence test is defined by running a simulation on a sequence of refined meshes k, see Fig. 7 to see three of them, and compare the numerical solution at the final time against an exact solution. Note that finer meshes are not self refinements of coarser meshes.

We start by fixing the order of the numerical method and run the simulation for each mesh. At final time we measure the error ($\mathcal{E}_k$) of the complete domain Ω comparing against an exact solution using a suitable norm, for example L_1, L_2 or L_∞. Comparing these errors for different mesh spacing (h_k) gives us the order of convergence of the numerical scheme via the expression

$$\mathcal{O} = \frac{\log(\mathcal{E}_{k+1}/\mathcal{E}_k)}{\log(h_{k+1}/h_k)}. \tag{32}$$

GMDD
6, 3743–3786, 2013

4.1 Convergence test 1 for advection equation: Constant coefficients

Here we test the convergence properties of the CUDA implementation solving the conservative form Eq. (1) applied to the linear advection equation. We scale the periodic domain to $\Omega = [-0.5, 0.5] \times [-0.5, 0.5]$ and define a constant speed setting $a_x(\boldsymbol{x}) = 1.0$ and $a_y(\boldsymbol{x}) = 1.0$, see Appendix C for details of the PDE. The final time is $t_{\text{end}} = 2.0$ such that the numerical solution is the same as the initial condition. The initial condition is defined using a Gaussian function

$$u(x, y) = a \exp\left(-\frac{(x - x_0)^2}{2\sigma_x^2} - \frac{(y - y_0)^2}{2\sigma_y^2} \right), \tag{33}$$

with $a = 0.2$, $(x_0, y_0) = (-0.2, -0.2)$ and $\sigma_x = \sigma_y = 0.05$. In Fig. 8 we plot the value of $u(x = 0.2, y = 0.2, t)$ in time for different meshes for the fourth order method. In black we have a reference solution computed with the seventh order method and mesh $h = 0.025$ (the right one in Fig. 7). We also show the difference between the reference solution and the numerical solution multiplied by a factor of 10, obtained with the fourth order method using the three meshes depicted in Fig. 7. We clearly see that the solution with mesh size $h = 0.05$ is already very good with an error of the order of 1 %.

In Fig. 9 we plot the error versus mesh spacing and computational time. We obtain the expected order of convergence up to seventh order (p6). We observe what seems to be a threshold around 10^{-11} where the error does not decrease any further as we increase the order of the scheme or refine the mesh. We also observe that for any given error level, higher order schemes are more efficient. The segmented line is the result from the CPU code. The errors measured for the CPU and GPU implementation are equivalent up to machine accuracy therefore superimposed on the left side of Fig. 9. On the right side we see a big difference between these two lines with the CPU implementation being one order of magnitude more expensive. However, from this comparison we can not conclude any speed up factor, since no special care was taken to optimise the CPU code. A discussion of this fact will be postponed to the conclusions section.

GMDD
6, 3743–3786, 2013

4.2 Convergence test 2 advection equation: Variable coefficients

This test problem has spatial dependent coefficient for the velocity field and a periodic domain scaled to $\Omega = [-0.5, 0.5] \times [-0.5, 0.5]$. The velocity field is defined using an auxiliary coordinate system (x', y') rotated in 45° counter clock wise from the original Cartesian coordinates. In this auxiliary coordinate system we define the velocity field by

$$v(x') = v^0 + \alpha \cos\left(\frac{2\pi \mathrm{d}(x')}{\lambda}\right), \quad \mathrm{d}(x') = |x'|, \tag{34}$$

with $v^0 = 1.0$, $\alpha = 0.2$, $\lambda = 2/\sqrt{2}$ and $d(x')$ the distance of a point to the y' axis. Then, this velocity is projected back to the Cartesian coordinates defining $a_x(\boldsymbol{x}) = v(x')\cos(\pi/4)$ and $a_y(\boldsymbol{x}) = v(x')\sin(\pi/4)$. In Fig. 10 we show the velocity field and the auxiliary coordinate system. The final time t_{end} is computed such that the solution coincides with the initial condition as

$$t_{\text{end}} = \int_0^{\sqrt{2}} \frac{1}{v(x')} \mathrm{d}x' = 1.443375673. \tag{35}$$

As in test problem 1 we use a Gaussian function (33) to define the initial condition using $a = 0.2$, $(x_0, y_0) = (0, 0)$ and $\sigma_x = \sigma_y = 0.05$. In Fig. 11 we show the convergence results. We observe the expected order of convergence up to a threshold error level (around 10^{-9} for the double precision results) where the error reach machine accuracy. This threshold is in the order of 10^{-5} for the single precision runs plotted with dashed lines (SP GPU p3m3 and SP GPU p4m4).

4.3 Convergence test 3 advection equation: A swirling deformation flow

The following test is adapted from (LeVeque, 1996) and consists of a velocity field that is space and time-dependent. The main idea is to have a swirling deformation flow that

GMDD
6, 3743–3786, 2013

after half the period changes direction to recover the initial condition described by the following functions

$$a_x(\boldsymbol{x}) = \sin(\pi x)^2 \sin(2\pi y) g(t),$$
$$b_x(\boldsymbol{x}) = -\sin(\pi y)^2 \sin(2\pi x) g(t). \tag{36}$$

Function $g(t) = \cos(\pi t/T)$ is responsible for the time dependency with $T = 1.5$ the period. The initial condition is a Gaussian function (33) with $a = 0.5$, $(x_0, y_0) = (0.75, 0.5)$ and $\sigma_x = \sigma_y = 0.05$ and the computational domain is the unit square. We use the same meshes as presented in Fig. 7. For this test we observe the expected convergence measured in the L_2 and L_∞ norm. In this case, as the wind field has space dependency we also required a high-order representation of matrix coefficients. We plot also in Fig. 7 the numerical solution obtained from a / semi-Lagrangian code (Behrens, 1996) with dashed line.

4.4 Convergence test elastic wave equation

This convergence test problem was obtained from Käser and Dumbser (2006). We use a square computational domain scaled to $\Omega = [-50, 50] \times [-50, 50]$ with periodical boundaries and evolve the initial condition for one period such that we can compare with the exact solution, in this case the initial condition. We run this test fixing the order of the method, for example second order (P1), over several meshes that were constructed with increasing number of elements (decreasing mesh size). Figure 14 depicts the results using the model presented in Sect. 2.3 obtained for the seismic wave equation. On the left we plotted error versus mesh size while on the right error against computing time. We compare single (SP) and double (DP) precision runs in the GPU code and the CPU code from SeisSol (2013). The solid lines are the double precision runs from second to sixth order method where we see that the expected convergence is reached (slope of the lines) and is equivalent to the error level obtained

GMDD
6, 3743–3786, 2013

with SeisSol (segmented lines). Note also that at small error levels, in particular the DP P5 line reaches a limit where it seems that no further improvements are possible. The same behaviour is observed for the SeisSol code but with a smaller error. Something similar happens for the single precision runs (dotted lines) where the errors behave as expected following the double precision results up to a threshold where the error remains the same even if we increase the order of the method or further refine the mesh. In the right part we plot the error level against the computational time. We can conclude that for a specific error, say 10^{-4}, a higher order method is more efficient than a lower order method as it requires less time to reach the same error level. We observe also that in the region where the single and double precision GPU code generate the same error level there is no speed up in the case of single precision. When comparing computational time between our GPU implementation and the CPU one we observe that the GPU is faster. This advantage decreases when higher order is used.

Visible is the overhead in a GPU. The time increases in particular when the work load is low (coarse meshes with low number of elements) and can lead even to higher run times than the CPU version. However, this overhead decreases relatively when more elements have to be computed so that the GPU is fully loaded and becomes more efficient than the CPU code.

5 Conclusions and future work

In this manuscript we presented a GPU implementation of the arbitrarily high-order ADER-DG numerical method considering unstructured meshes in two spatial dimensions. We presented a strategy to adapt memory allocation to the specific hardware. The implemented code can be easily adapted to other linear hyperbolic equations by using the CUDA-C library approach.

We found that the expected order of accuracy is reached for the presented test cases. We also observe that there is a threshold after which the error can not be reduced. This is in particular evident when running the GPU code in single precision mode. For

GMDD
6, 3743–3786, 2013

computations where very small errors are required, we suggest to use only double precision implementations. With respect to computational time, we observe that the GPU code give us a speed up factor comparing against a CPU implementation. The reduced computational time can be assigned to several components, for example the specialized hardware architecture of the GPUs, the implementation effort to optimize the code and the benchmark used to measure it. With respect to the latter one we must keep in mind that a CPU implementation is subject to optimizations as well and therefore the relative (CPU vs GPU) speed up strongly depends on how much effort was used to code it. Finally we note that during the GPU algorithm development and implementation some limited optimizations were necessary due to hardware constraints and to respect available resources of the graphic cards.

As a future work we are considering to implement a parallel GPU simulation and extend this implementation to three spatial dimensions.

Appendix A

Reference coordinate system

We make use of a reference coordinate system (ξ, η) where the basis functions are defined using the Jacobian polynomials. If triangular elements are used the reference element E^{R} is defined by $0 \leq \xi \leq 1$ and $0 \leq \eta \leq 1-\xi$. If quadrangular elements are used the reference element E^{R} is defined by $0 \leq \xi \leq 1$ and $0 \leq \eta \leq 1$.

The mapping between the physical (x, y) and the reference (ξ, η) coordinate system is represented by the Jacobian matrix $J = \partial(x, y)/\partial(\xi, \eta)$.

$$J = \begin{bmatrix} x_\xi & x_\eta \\ y_\xi & y_\eta \end{bmatrix}, \quad J^{-1} = \begin{bmatrix} \xi_x & \xi_y \\ \eta_x & \eta_y \end{bmatrix}, \tag{A1}$$

GMDD
6, 3743–3786, 2013

Naming the Del operator in Cartesian coordinates $\nabla = [\partial_x, \partial_y]$ and $\nabla^{\xi} = [\partial_{\xi}, \partial_{\eta}]$ in the reference coordinates we can write

$$\nabla = \nabla^{\xi} J^{-1} \tag{A2}$$

For triangular elements J and its determinant $|J|$ are constant. On the other hand on quadrangular elements J and $|J|$ are constant only for parallelograms.

Appendix B

L_2 projection

We use the L_2 projection at time $t = 0$ to project he initial condition into the time-dependent degrees of freedom $\hat{u}^i_{pl}(t = 0)$ that represent the numerical solution inside element E^i, where $\boldsymbol{x} \in E^i$.

$$\hat{u}^i_{pl}(t = 0) = \frac{\int_{E^R} u_p(\boldsymbol{x}(\xi, \eta), 0)\, \phi_l(\xi, \eta)\, \mathrm{d}\xi \mathrm{d}\eta}{M_{ll}}. \tag{B1}$$

The number of dof used depend on the order of the numerical method that we want to use with $l = 1, \ldots, N_d$.

We project the Jacobian matrices A_{pq} and B_{pq} in the same manner. This is done once at the beginning of the computation for each element E^i and stored. The order of the approximation of these matrices is independent of the order of the numerical

GMDD
6, 3743–3786, 2013

method therefore we use $m = 1, \ldots, N_{dm}$ basis functions. For $\boldsymbol{x} \in E^i$ we use

$$\hat{A}^i_{pqm} = \frac{\int_{E^R} A_{pq}(\boldsymbol{x}(\xi,\eta))\,\phi_m(\xi,\eta)\,\mathrm{d}\xi\mathrm{d}\eta}{M_{mm}}, \qquad \hat{B}^i_{pqm} = \frac{\int_{E^R} B_{pq}(\boldsymbol{x}(\xi,\eta))\,\phi_m(\xi,\eta)\,\mathrm{d}\xi\mathrm{d}\eta}{M_{mm}}. \tag{B2}$$

The star matrices defined in the volume integral (10) are constructed from $\hat{A}^i_{pqm}$ and $\hat{B}^i_{pqm}$ as follows

$$\hat{A}^{*i}_{pqm} = \xi_x \hat{A}^i_{pqm} + \xi_y \hat{B}^i_{pqm}, \qquad \hat{B}^{*i}_{pqm} = \eta_x \hat{A}^i_{pqm} + \eta_y \hat{B}^i_{pqm}. \tag{B3}$$

We remark that in this work we restrict to mesh topologies where the Jacobian mapping matrix **J** is constant inside each element. In practice this means that we can use triangular and quadrangular elements with the later at most need to be parallelograms.

Appendix C

Partial differential equations

For completeness, here we show the two partial differential equations used to test the ADER-DG algorithm.

GMDD
6, 3743–3786, 2013

C1 The advection equation

The advection equation is an hyperbolic scalar partial differential equation that can represent the propagation of a tracer driven by a velocity field, see Chapter 2.1 of LeVeque (2002).

$$\frac{\partial u(\boldsymbol{x},t)}{\partial t} + \frac{\partial f(\boldsymbol{x},t)}{\partial t} + \frac{\partial g(\boldsymbol{x},t)}{\partial t} = 0. \tag{C1}$$

with $f(\boldsymbol{x},t) = a_x(\boldsymbol{x})u(\boldsymbol{x},t)$ and $g(\boldsymbol{x},t) = a_y(\boldsymbol{x})u(\boldsymbol{x},t)$. The space dependent velocity field is described by its two components $a_x(\boldsymbol{x})$ and $a_y(\boldsymbol{x})$. In this case, when the velocity is space dependent we can refer to the variable coefficient linear equation, see Chapter 9 of LeVeque (2002).

C2 The elastic wave equation

The elastic wave equation is an hyperbolic partial differential equation that can represent the propagation of seismic waves in an elastic medium. In its velocity-stress formulation reads,

$$\frac{\partial \mathbf{U}(\boldsymbol{x},t)}{\partial t} + \mathbf{A}(\boldsymbol{x})\frac{\partial \mathbf{U}(\boldsymbol{x},t)}{\partial x} + \mathbf{B}(\boldsymbol{x})\frac{\partial \mathbf{U}(\boldsymbol{x},t)}{\partial y} = 0. \tag{C2}$$

Matrices **A** and **B** are the Jacobian matrices and describe the elastic medium where the waves propagate. Vector $\mathbf{U} = [\sigma_{xx}, \sigma_{yy}, \sigma_{xy}, u_x, u_y]^T$ is the unknown vector corresponding to the normal stress in x and y direction, the shear stress and the particle

GMDD
6, 3743–3786, 2013

velocities in x and y direction. The Jacobian matrices are

$$\mathbf{A} = \begin{pmatrix} 0 & 0 & 0 & -(\lambda+2\mu) & 0 \\ 0 & 0 & 0 & -\lambda & 0 \\ 0 & 0 & 0 & 0 & -\mu \\ -\frac{1}{\rho} & 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{1}{\rho} & 0 & 0 \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} 0 & 0 & 0 & 0 & -\lambda \\ 0 & 0 & 0 & 0 & -(\lambda+2\mu) \\ 0 & 0 & 0 & -\mu & 0 \\ 0 & 0 & -\frac{1}{\rho} & 0 & 0 \\ 0 & -\frac{1}{\rho} & 0 & 0 & 0 \end{pmatrix}. \tag{C3}$$

with ρ the density and λ and μ the Lamé constants. See Käser and Dumbser (2006); Castro et al. (2010) for further details.

Acknowledgements. This work was supported through the Cluster of Excellence "CliSAP" (EXC177), University of Hamburg, funded through the German Science Foundation (DFG). The co-authors acknowledge funding through Volkswagen Foundation. The first author also thanks Oliver Kunst for sharing his helpful knowledge on the C++ language and all its mysteries. Finally, big thanks to the Numerical Methods in Geosciences group in KlimaCampus. Thanks also to Alexander Breuer for his comments helping us to improve this work.

References

Asouti, V. G., Trompoukis, X. S., Kampolis, I. C., and Giannakoglou, K. C.: Unsteady CFD computations using vertex-centered finite volumes for unstructured grids on Graphics Processing Units, Int. J. Numer. Meth. Fl., 67, 232–246, 2011. 3745

Behrens, J.: An Adaptive Semi-Lagrangian Advection Scheme and its Parallelization, Mon. Wea. Rev., 124, 2386–2395, 1996. 3764

GMDD

6, 3743–3786, 2013

Brodtkorb, A. R., Sætra, M. L., and Altinakar, M.: Efficient shallow water simulations on GPUs: Implementation, visualization, verification, and validation, Comput. Fluids, 55, 1–12, 2012. 3745

Castro, C. E., Käser, M., and Brietzke, G. B.: Seismic waves in heterogeneous material: subcell resolution of the discontinuous Galerkin method, Geophys. J. Int., 182, 250–264, 2010. 3745, 3747, 3749, 3756, 3770

de la Asunción, M., Castro, M. J., Fernández-Nieto, E., Mantas, J. M., Acosta, S. O., and González-Vida, J. M.: Efficient GPU implementation of a two waves TVD-WAF method for the two-dimensional one layer shallow water system on structured meshes, Comput. Fluids, 80, 441–452, 2012. 3745

Dumbser, M.: Arbitrary High Order Schemes for the Solution of Hyperbolic Conservation Laws in Complex Domains, Ph.D. thesis, Universität Stuttgart, Institut für Aerodynamik und Gasdynamik, 2005. 3747

Hermann, V., Käser, M., and Castro, C. E.: Non-conforming hybrid meshes for efficient 2-D wave propagation using the Discontinuous Galerkin Method, Geophys. J. Int., 184, 746–758, 2011. 3747

Käser, M. and Dumbser, M.: An arbitrary high-order discontinuous Galerkin method for elastic waves on unstructured meshes I. The two-dimensional isotropic case with external source terms, Geophys. J. Int., 166, 855–877, 2006. 3745, 3747, 3756, 3764, 3770

Komatitsch, D., Erlebacher, G., Gödeke, D., and Michéa, D.: High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster, J. Comput. Phys., 229, 7692–7714, 2010. 3745

LeVeque, R. J.: High-resolution conservative algorithms for advection in incompressible flow, SIAM J. Numer. Anal., 33, 627–665, 1996. 3763

LeVeque, R. J.: Finite volume methods for hyperbolic problems, Cambridge, 2002. 3756, 3769

Meuer, H., Strohmaier, E., Dongarra, J., and Simon, H.: TOP500 Supercomputer sites, available at: http://www.top500.org/list/2012/11/ (last access: November 2012), 2012. 3745

Mu, D., Chen, P., and Wang, L.: Accelerating the discontinuous Galerkin method for seismic wave propagation simulations using the graphic processing unit (GPU) single-GPU implementation, Comput. Geosci., 51, 282–292, 2013. 3745

NVidia: CUDA C Best Practices Guide, available at: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html, last access: November 2012a. 3759

GMDD
6, 3743–3786, 2013

NVidia: CUDA C Programming Guide, available at: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, last access: November 2012b. 3758
Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., and Purcell, T. J.: A Survey of General-Purpose Computation on Graphics Hardware, Comput. Graph Forum, 26, 80–113, 2007. 3744
Pelties, C., Käser, M., Hermann, V., and Castro, C. E.: Regular versus irregular meshing for complicated models and their effect on synthetic seismograms, Geophys. J. Int., 183, 1031–1051, 2010. 3747
Pelties, C., de la Puente, J., Ampuero, J.-P., Brietzke, G. B., and Käser, M.: Three-dimensional dynamic rupture simulation with a high-order discontinuous Galerkin method on unstructured tetrahedral meshes, J. Geophys. Res., 117, B02309, doi:10.1029/2011JB008857, 2012. 3747
Rietmann, M., Messmer, P., Nissen-Meyer, T., Peter, D., Basini, P., Komatitsch, D., Schenk, O., Tromp, J., Boschi, L., and Giardini, D.: Forward and Adjoint Simulations of Seismic Wave Propagation on Emerging Large-Scale GPU Architectures, sC12, 10–16 November, Salt Lake City, Utah, USA, 2012. 3745
Rusanov, V. V.: On difference schemes of third-order accuracy for nonlinear hyperbolic systems, J. Comput. Phys., 5, 507–516, 1970. 3751
SeisSol: The SeisSol working group, available at: http://seissol.geophysik.uni-muenchen.de/ (last access: May 2013), 2013. 3747, 3764
Virieux, J.: SH-wave propagation in heterogeneous media: Velocity-stress finite-difference method, Geophysics, 49, 1933–1957, 1984. 3756
Virieux, J.: P-SV wave propagation in heterogeneous media: Velocity-stress finite-difference method, Geophysics, 51, 889–901, 1986. 3756

GMDD
6, 3743–3786, 2013

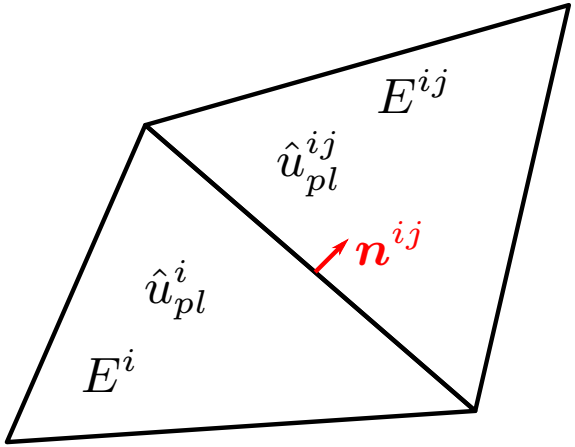

Fig. 1. Element E^i and its direct j-th neighbour E^{ij}. The normal vector $\boldsymbol{n}^{ij}$ is defined pointing outward from the j-th edge of element E^i.

GMDD
6, 3743–3786, 2013

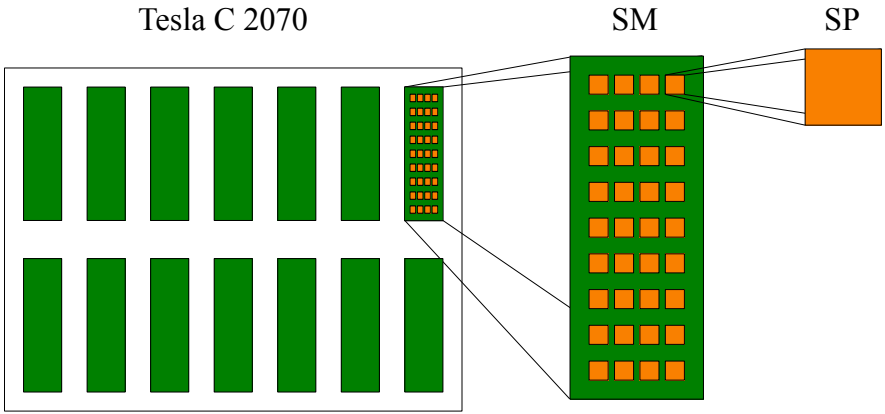

Fig. 2. Hardware distribution of the CUDA cores in a Tesla C2070 graphic card.

GMDD
6, 3743–3786, 2013

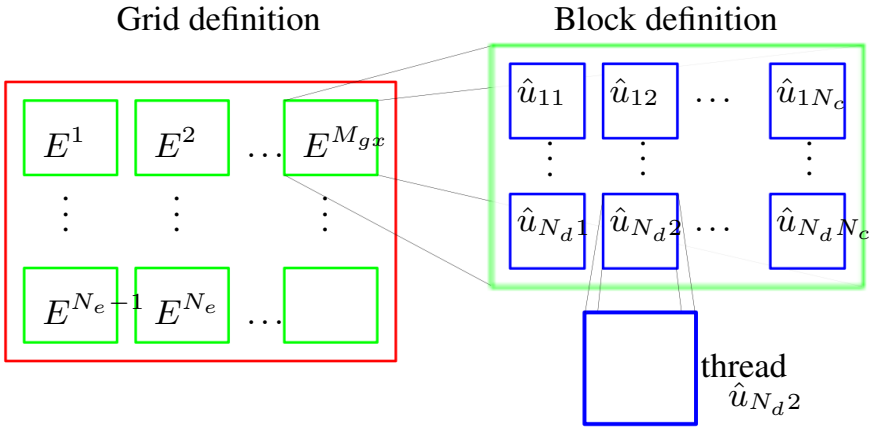

Fig. 3. Grid and block definition. We call M_{gx} the maximum x direction block number which depends on the GPU used. Here we show the mapping between blocks and elements of our mesh E^i where one block is assigned to one element. In the same manner each thread block is assigned to one of the degrees of freedom from E^i.

GMDD
6, 3743–3786, 2013

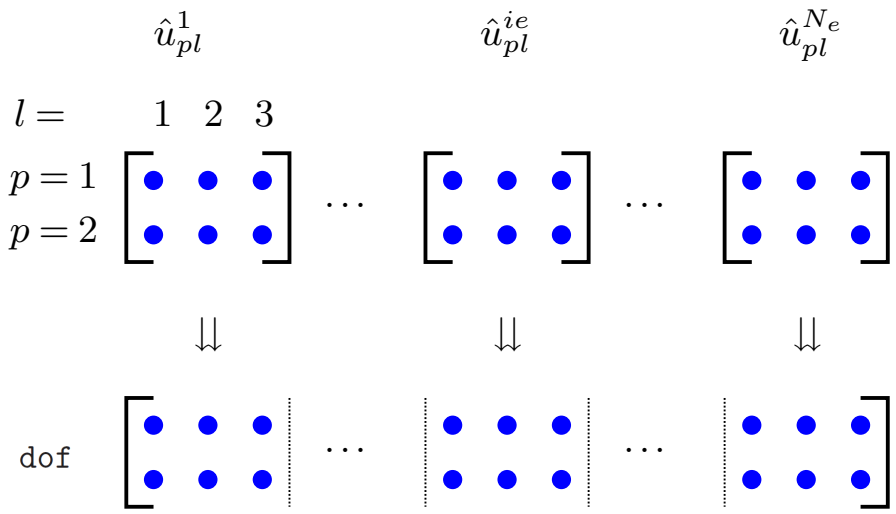

Fig. 4. Construction of `dof` to store the degrees of freedom of all elements in one single array inside GPU global memory. Here we see the example for a pde with 2 components and 3 degrees of freedom to approximate vector **U**.

GMDD
6, 3743–3786, 2013


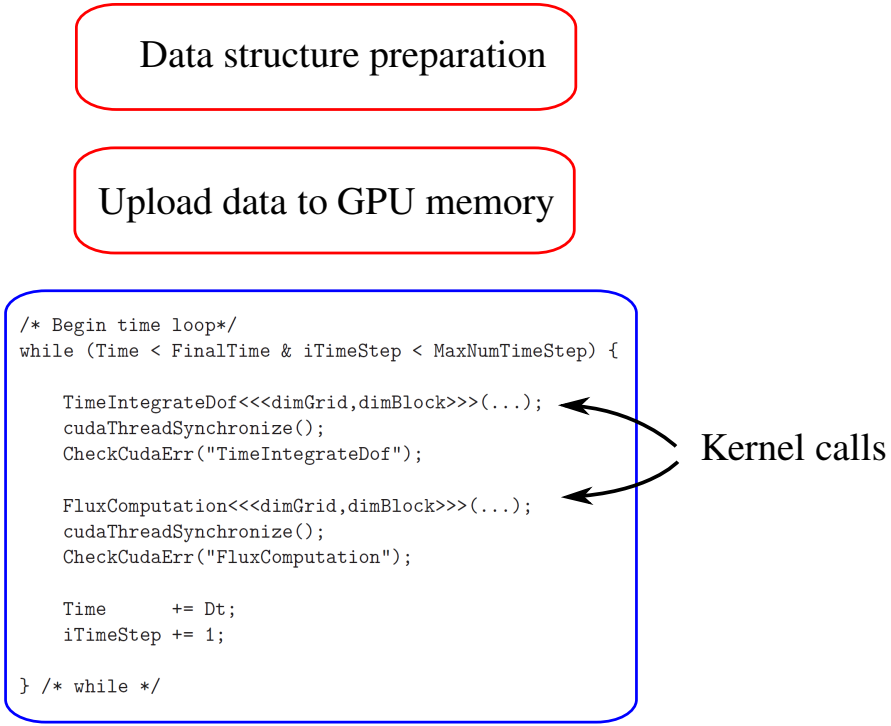

Fig. 5. Schematic view of the CUDA-C driver function that prepares and uploads the data into the GPU memory before running the time loop.

GMDD
6, 3743–3786, 2013

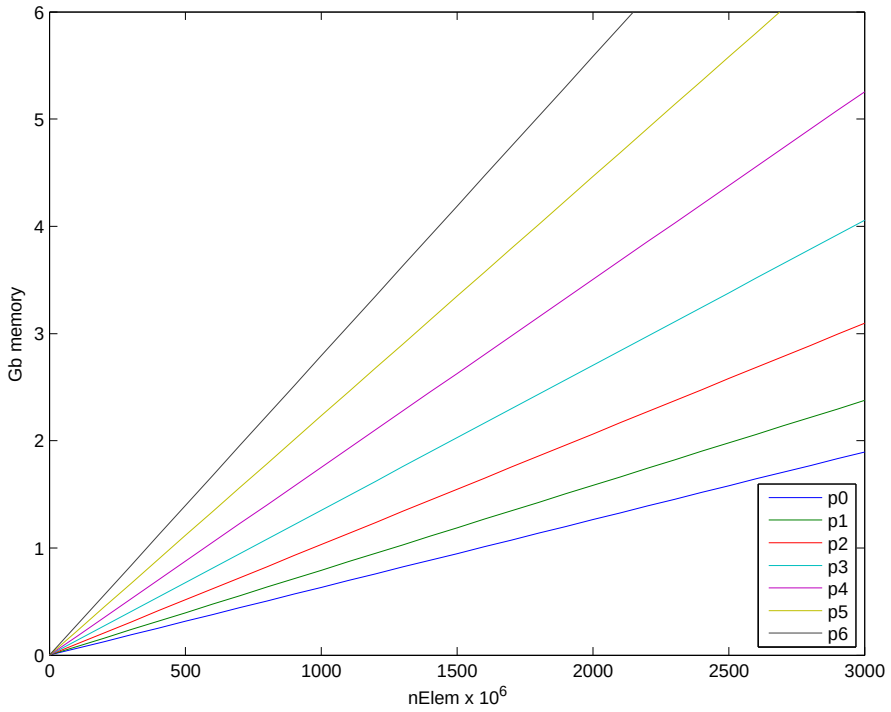

Fig. 6. Here we see the relation between memory requirements versus accuracy order of the numerical method presented in this manuscript considering a 5 component PDE and double precision representation for real numbers.

GMDD
6, 3743–3786, 2013

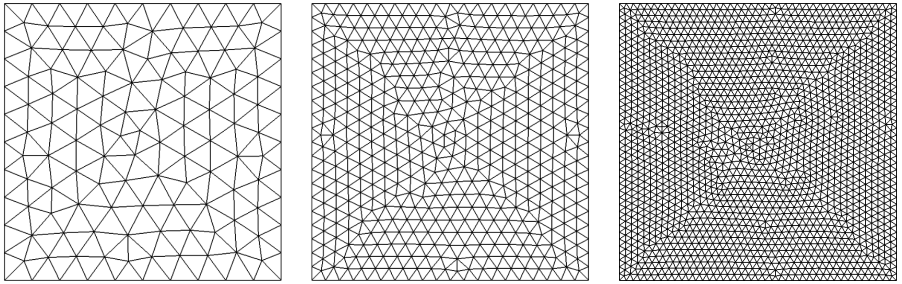

Fig. 7. Meshes used in the convergence test. From left to right we see three meshes with characteristic length $h = 0.1$, $h = 0.05$ and $h = 0.025$.

GMDD
6, 3743–3786, 2013

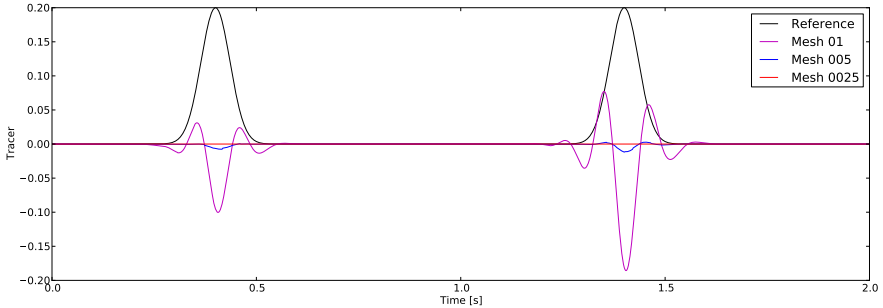

Fig. 8. Time signal for test 1. The black line is the reference solution obtained with mesh 0.025 and seventh order method. The purple, blue and red lines were obtained using the fourth order method and we plot the difference against the reference solution, multiplied by a factor of 10.

GMDD
6, 3743–3786, 2013

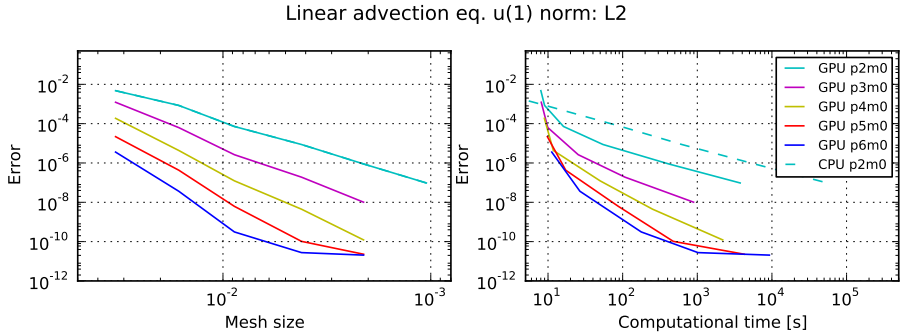

Fig. 9. Convergence test 1 for linear advection equation. The numerical solutions are named with the order of the method p2 to p6 while m0 represents the approximation order to the background field, in this case is constant.

GMDD
6, 3743–3786, 2013

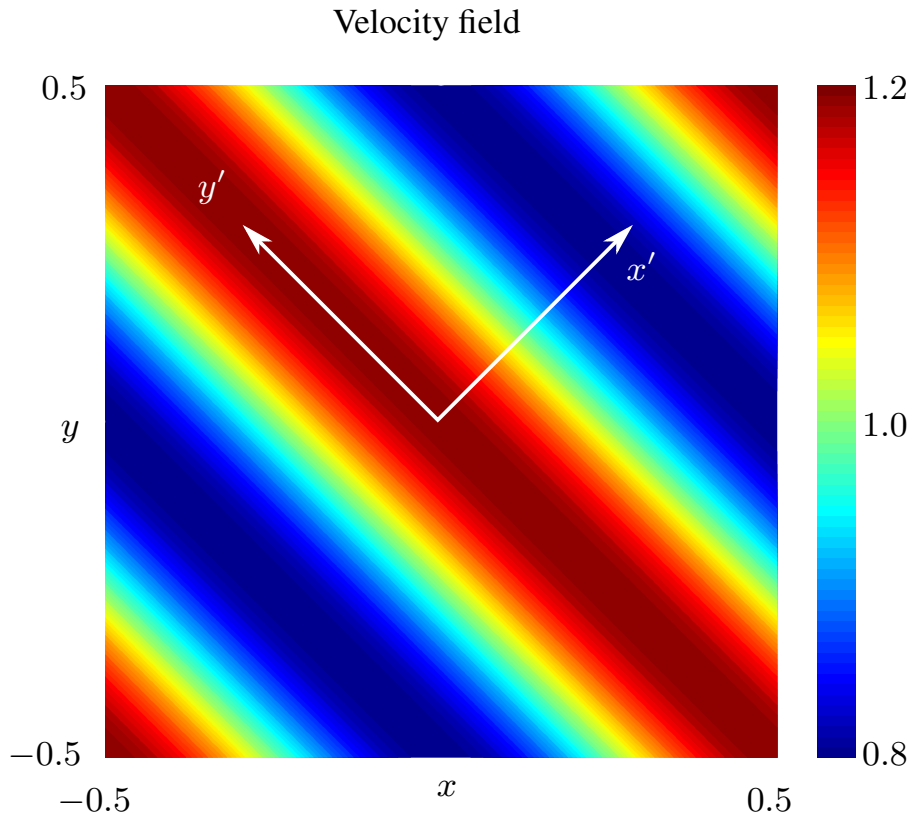

Fig. 10. Velocity field for the convergence test 2 including the auxiliary coordinate system (x', y').

GMDD
6, 3743–3786, 2013

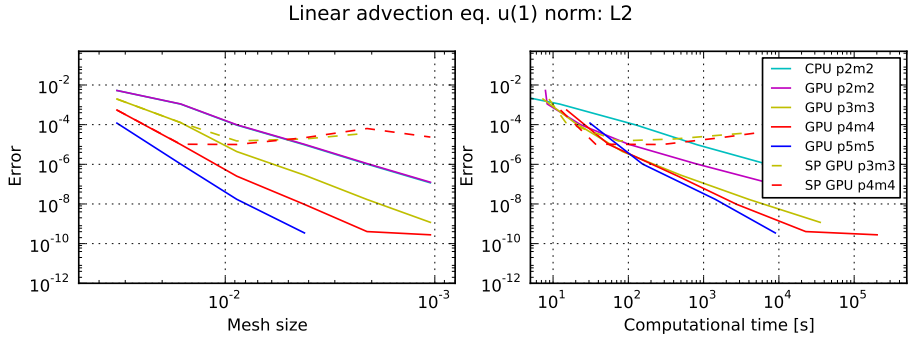

Fig. 11. Convergence test 2 for tracer advection equation. On the left we plot the error level versus the mesh size for L_2 norm. We observe the expected convergence for the numerical method of order 3 (p2m2) to 6 (p5m5) using double precision. We show also two runs obtained with single precision which hit machine accuracy much earlier than the double precision runs.

GMDD
6, 3743–3786, 2013

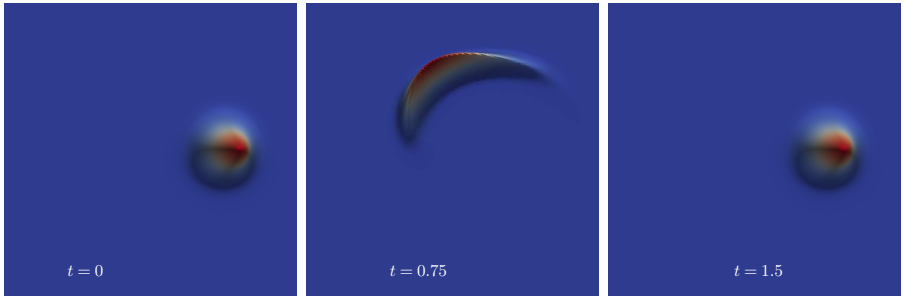

Fig. 12. Snapshot of the numerical solution test 3. At time $t = 0$ the initial condition begins to deform due to the flow field. At time $t = 0.75$ reaches the maximum deformation and the velocity flow change direction to recover the initial condition at time $t = 1.5$

GMDD
6, 3743–3786, 2013

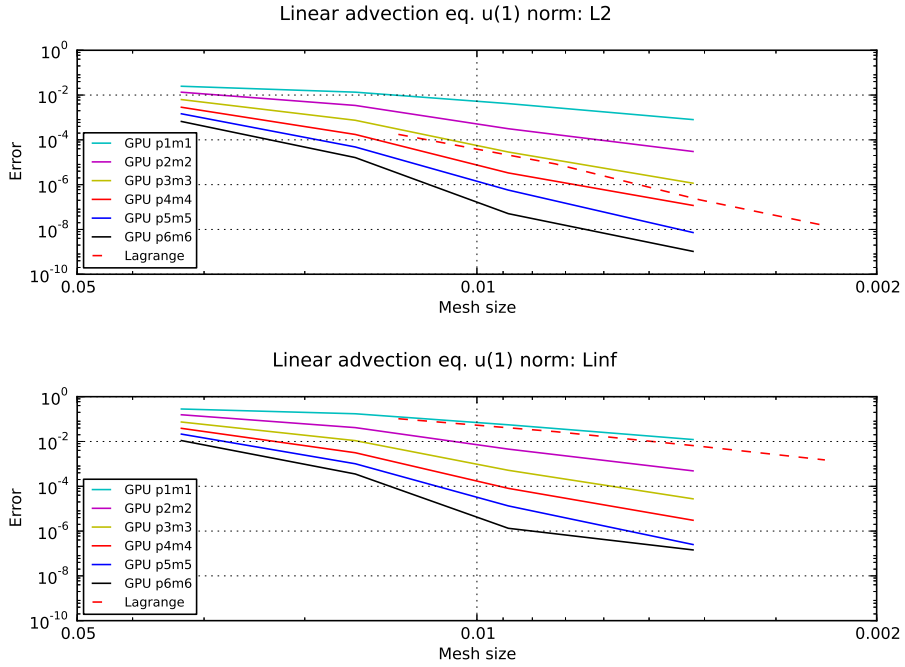

Fig. 13. Convergence Test 3 for the tracer advection equation. We plot the L_2 and L_∞ norm.

GMDD
6, 3743–3786, 2013

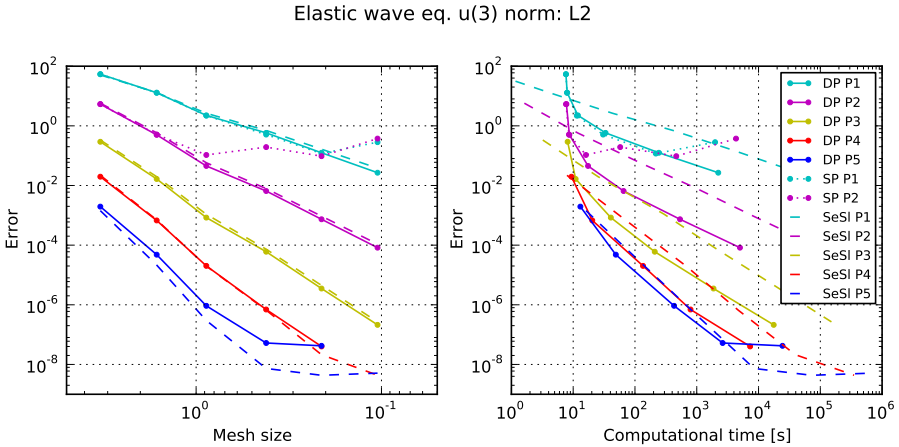

Fig. 14. Convergence test of the ADER-DG code applied to the linear elastic wave equation. We compare double precision (DP) and single precision (SP) of the GPU implementation against the CPU code SeisSol (SeSl) for different order from second (P1) to sixth (P5). In the vertical axis the error level obtained using the L_2 norm. The figure on the left depicts the error against mesh size while on the right the horizontal axis represents computational time.

GMDD
6, 3743–3786, 2013