2017

A GPU-accelerated continuous and discontinuoous Galerkin non-hydrostatic atmospheric model

Abdi, Daniel S.; Wilcox, Lucas C.; Warburton, Timothy C.; Giraldo, Francis X.

Sage Publishing

D.S. Abdi, L.C. Wilcox, T.C. Warburton, F.X. Giraldo, "A GPU-sccelerated continuous and discontinuous Galerkin non-hydrostatic atmospheric model," International Journal of High Performance Computing Applications, (2017), pp. 1-29
https://hdl.handle.net/10945/56094

Full Length Article

International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
1–29

DOI: 10.1177/1094342017694427
journals.sagepub.com/home/hpc
SAGE

A GPU-accelerated continuous and discontinuous Galerkin non-hydrostatic atmospheric model

Daniel S Abdi[1], Lucas C Wilcox[1], Timothy C Warburton[2] and Francis X Giraldo[1]

Abstract

We present a Graphics Processing Unit (GPU)-accelerated nodal discontinuous Galerkin method for the solution of the three-dimensional Euler equations that govern the motion and thermodynamic state of the atmosphere. Acceleration of the dynamical core of atmospheric models plays an important practical role in not only getting daily forecasts faster, but also in obtaining more accurate (high resolution) results within a given simulation time limit. We use algorithms suitable for the single instruction multiple thread architecture of GPUs to accelerate our model by two orders of magnitude relative to one core of a CPU. Tests on one node of the Titan supercomputer show a speedup of up to 15 times using the K20X GPU as compared to that on the 16-core AMD Opteron CPU. The scalability of the multi-GPU implementation is tested using 16,384 GPUs, which resulted in a weak scaling efficiency of about 90%. Finally, the accuracy and performance of our GPU implementation is verified using several benchmark problems representative of different scales of atmospheric dynamics.

Keywords

NUMA, GPU, HPC, OCCA, atmospheric model, discontinuous Galerkin, continuous Galerkin

1 Introduction

Most operational Numerical Weather Prediction (NWP) models are based on the finite difference or spectral transform spatial discretization methods. *Finite difference* methods are popular with limited area models, due to their ease of implementation and good performance on structured grids, whereas global circulation models mostly use the spectral transform method. *Spectral transform methods* often do not scale well on massively parallel systems due to the need for global (*all-to-all*) communication required by the Fourier transform. On the other hand, the finite difference method requires wide halo layers at inter-processor boundaries to achieve high-order accuracy.The search for efficient parallel NWP codes in the era of high-performance computing suggests the use of alternative methods that have local operation properties while still offering high-order accuracy (Nair et al., 2011); their efficiency coming from the minimal parallel communication footprint that is of vital importance as resolution increases. The Non-hydrostatic Unified Model of the Atmosphere (NUMA) is one such NWP model that offers high-order accuracy while using local methods for parallel efficiency (Giraldo and Rosmond, 2004; Giraldo and Restelli, 2008; Kelly and Giraldo, 2012; Marras et al., 2015).

In Table 1, we give a summary of a recent review on the progress of porting several NWP models to the GPU (Sawyer, 2014). Among those models that ported the whole dynamical core, a maximum overall speedup of 3 times (from here on, we shall use, e.g. $3\times$ to represent such a speedup) is observed for a GPU relative to a multi-core CPU. The only spectral element model in the review was the Community Atmospheric Model (CAM-SE) that showed a speedup of $3\times$ for the dynamical core using CUDA.

When comparing the acceleration of CAM-SE tracer kernels using OpenACC, though substantially easier to program than the CUDA version, they performed $1.5\times$ slower (Norman et al., 2015). This could occur,

[1]Department of Applied Mathematics, Naval Postgraduate School, USA
[2]Department of Mathematics, Virginia Tech University, USA

Corresponding author:
Daniel S Abdi, Naval Postgraduate School, Monterey, CA 93943, USA.
Email: dsabdi@nps.edu

Table 1 GPU acceleration of a few atmospheric models based on a summary in sawyer (2014). The only spectral element (SE) code is the hydrostatic CAM-SE model. A maximum speedup of 3× over a multi-core CPU is observed among those models that have ported the whole dynamical core.

Model	Non-hydrostatic	Method	GPU ported	Speedup	Language
CAM-SE	No	SE	Parts of DyCore	3×	CUDA + OpenACC
WRF	Yes	FD	Parts of DyCore	2×	CUDA + OpenACC
NICAM	Yes	FV	DyCore	3×	OpenACC
ICON	Yes	FV	DyCore	2×	CUDA + OpenACC + OpenCL
GEOS-5	Yes	FV	Parts of DyCore	5×	CUDA + OpenACC
FIM/NIM	Yes	FV	DyCore + Physics	3×	F2C-ACC + OpenACC
GRAPES	Yes	SL	Parts of DyCore	4×	CUDA
COSMO	Yes	FD	DyCore + Physics	2×	CUDA + OpenACC

for example, by not fully exploiting the private worker array capability of OpenACC. The most important metric we shall use to compare performance on the GPU is speedup, however, we should note that speedup results are significantly influenced by how well the CPU and GPU codes are optimized. For this reason, we shall also report individual GPU kernel performance in terms of rate of floating point operations and rate of data transfer (*bandwidth*) and will illustrate our results using roofline models.

Element-based Galerkin (EBG) methods, in which the basis functions are defined within an element, are well suited for distributed computing for two reasons (Klöckner et al., 2009): Firstly, localized memory accesses result in low communication overhead. In contrast, global methods require an *all-to-all* communication that severely degrades scalability on most architectures and methods having non-compact high-order support require larger halo regions, which translates to larger communication stencils that also reduce scalability. Secondly, high-order polynomial expansion of the solution results in a large arithmetic intensity per degree of freedom. These two properties work in favor of EBG methods for Graphic Processing Unit (GPU) computing as well. The two EBG methods of NUMA, namely continuous Galerkin (CG) and discontinuous Galerkin (DG), are ported to the GPU in a unified manner (see section 3.3). Parallel implementation of DG is often easier and more efficient than that of CG because of a smaller communication stencil; with a judicious choice of numerical flux, only neighbors sharing a face need to communicate in DG, as opposed to the edge and corner neighbor communication required by CG. Moreover, DG allows for a simple overlap of computation of volume integrals and intra-processor flux with communication of boundary data, which can be exploited to improve the efficiency of the parallel implementation (Kelly and Giraldo, 2012). CG can also benefit from a communication–computation overlap, but it requires a bit more work than that for DG (Deville et al., 2002).

EBG methods have been successfully ported to GPUs to speed up the solution of various partial differential equations (PDEs) by orders of magnitude. Acceleration of a CG simulation using GPUs was first reported by Goddeke et al. (2005). Later, Klöckner et al. (2009) made the first GPU implementation of nodal DG for the solution of linear hyperbolic conservation laws. They mention that non-trivial adjustments to the DG method are required to solve non-linear hyperbolic equations, such as the compressible Euler equations, on the GPU due to complexity of implementing limiters and artificial viscosity. Another notable difference with the current work is that NUMA uses a tensor-product approach with hexahedra elements for efficiency reasons (Kelly and Giraldo 2012). Klöckner et al. (2009) argue tetrahedra are preferable on the GPU due to larger arithmetic intensity and reduced memory fetches. More recently Siebenborn et al. (2012) implemented the Runge–Kutta (RK) discontinuous Galerkin method of Cockburn and Shu (1998) on the GPU to solve the non-linear Euler equations using tetrahedral grids. They reported a speedup of 18× over the serial implementation of the method running on a single-core CPU. Fuhry et al. (2014) made an implementation of the 2D discontinuous Galerkin on the GPU using triangular elements and obtained a speedup of about 50× relative to a single-core CPU. The approach they used is a one-element-per-thread strategy that is different from the one-node-per-thread strategy we shall use in this work when running on the GPU. However, thanks to our use of a device-agnostic language, the same kernel code used on the GPU, switches to using the one-element-per-thread strategy of Fuhry et al. (2014) when running on the CPU using OpenMP mode.Chan et al. (2015) presented a GPU acceleration of DG methods for the solution of the acoustic wave equation on hex-dominant hybrid meshes consisting of hexahedra, tetrahedra, wedges and pyramids. They mention that the DG spectral element formulation on hexahedra is more efficient on the GPU using Legendre–Gauss–Lobatto (LGL) points than using Gauss–Legendre (GL) points. To avoid the cost of storing the inverse mass matrix on the GPU, they used different basis functions that yield a diagonal mass matrix for each of the cell shapes except tetrahedra.

For straight-edged elements, the mass matrix for tetrahedral elements is not diagonal, but a scalar multiple of that of the reference tetrahedron, therefore the storage cost is minimal. In Chan and Warburton (2015), they consider the use of the Bernstein–Bezier polynomial basis for DG on the GPU to enhance the sparsity of the derivative and lift matrices as compared to classical DG with Lagrange polynomial basis. However, this comes at a cost of increased condition number of the matrices that could potentially cause stability issues. They conclude that, at high-order polynomial approximation, DG implemented with the Bernstein–Bezier polynomial basis perform better than a straightforward implementation of classical DG. Remacle et al. (2015) studied GPU acceleration of spectral elements for the solution of the Poisson problem on purely hexahedral grids. The solution of elliptic problems is most efficiently done using implicit methods; thus, they implemented a matrix-free Pre-conditioned Conjugate Gradient (PCG) on the GPU and demonstrated that problems with 50 million grid cells can be solved in a few seconds.

General purpose computing on GPUs can be done using several programming models from various vendors: AMD's OpenCL, NVIDIA's CUDA and OpenACC, to name a few. The choice of the programming model for a project depends on several factors. The goal of the current work is to port NUMA to heterogeneous computing environments in a performance-portable way, and hence cross-platform portability is the topmost priority. In the future we shall address performance portability using automatic code transformation techniques, such as Loo.py Klöckner and Warburton, 2013). To achieve cross-platform portability, we chose a new threading language called OCCA (Open Concurrent Compute Abstraction) (Medina et al., 2014), which is a unified approach to multi-threading languages. Kernels written in OCCA are cross-compiled at runtime to existing thread models such as OpenCL, CUDA, OpenMP, etc.; here, we present results only for OpenCL and CUDA backends and postpone OpenMP for future work. OCCA has been shown to deliver portable high performance for various EBG methods (Medina et al., 2014). It has already been used in Gandham et al. (2014) to accelerate the DG solution of the shallow water equations, in Remacle et al. (2015) for the Poisson problem, and in Modave et al. (2016) for acoustic and elastic problems.

2 Governing equations

The dynamics of non-hydrostatic atmospheric processes are governed by the compressible Euler equations. The equation sets can be written in various conservative and non-conservative forms. Among those, a conservative set is selected with the prognostic variables $(\rho, \mathbf{U}, \Theta)^\top$, where ρ is density, $\mathbf{U} = (U, V, W) = \rho\mathbf{u}$, $\Theta = \rho\theta$, where θ is potential temperature and $\mathbf{u} = (u, v, w)$ are the velocity components. We write the governing equations in the following way

$$\begin{aligned} &\frac{\partial \rho}{\partial t} + \nabla \cdot \mathbf{U} = 0 \\ &\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot \left(\frac{\mathbf{U} \otimes \mathbf{U}}{\rho} + P\mathbf{I}_3 \right) = -\rho \mathbf{g} \\ &\frac{\partial \Theta}{\partial t} + \nabla \cdot \left(\frac{\Theta \mathbf{U}}{\rho} \right) = 0 \end{aligned} \tag{1}$$

where $\mathbf{g}$ is the gravity vector.[1] The pressure in the momentum equation is obtained from the equation of state

$$P = P_0 \left(\frac{R\Theta}{P_0} \right)^{\gamma} \tag{2}$$

where $R = c_p - c_v$ and $\gamma = \frac{c_p}{c_v}$ for given specific heat of pressure and volume of c_p and c_v, respectively. We have selected to use a conservative form of the equations, to take advantage of not only global but also local conservation properties (given the proper discretization method).

For better numerical stability, the density, pressure and potential temperature variables are split into background and perturbation components. The background component is time-invariant and is often obtained by assuming hydrostatic equilibrium and a neutral atmosphere. Let us define the decomposition as follows

$$\begin{aligned} \rho(\mathbf{x}, t) &= \bar{\rho}(\mathbf{x}) + \rho'(\mathbf{x}, t) \\ \Theta(\mathbf{x}, t) &= \bar{\Theta}(\mathbf{x}) + \Theta'(\mathbf{x}, t) \\ P(\mathbf{x}, t) &= \bar{P}(\mathbf{x}) + P'(\mathbf{x}, t) \end{aligned}$$

where (x, t) are the space–time coordinates. Then, the modified equation set is

$$\begin{aligned} &\frac{\partial \rho'}{\partial t} + \nabla \cdot \mathbf{U} = 0 \\ &\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot \left(\frac{\mathbf{U} \otimes \mathbf{U}}{\rho} + P'\mathbf{I}_3 \right) = -\rho' \mathbf{g} \\ &\frac{\partial \Theta'}{\partial t} + \nabla \cdot \left(\frac{\Theta \mathbf{U}}{\rho} \right) = 0 \end{aligned} \tag{3}$$

In compact vector notation form

$$\frac{\partial \mathbf{q}}{\partial t} + \nabla . \mathcal{F}(\mathbf{q}) = \mathcal{S}(\mathbf{q}) \tag{4}$$

where $\mathbf{q} = (\rho', \mathrm{U}, \Theta')^\top$ is the solution vector, is $\mathcal{F}(\mathbf{q}) = (\mathbf{U}, \frac{\mathbf{U} \otimes \mathbf{U}}{\rho} + P'\mathbf{I}_3, \frac{\Theta \mathbf{U}}{\rho})^\top$ the flux vector, and $\mathcal{S}(\mathbf{q}) = (0, -\rho'\mathbf{g}, 0)^\top$ is the source vector.

For the purpose of stabilization, we add artificial viscosity to the governing equations as follows

$$\frac{\partial \mathbf{q}}{\partial t} + \nabla \cdot \mathcal{F}(\mathbf{q}) = \mathcal{S}(\mathbf{q}) + \nabla \cdot (\mu \nabla \mathbf{q}) \tag{5}$$

where μ is the constant artificial kinematic viscosity. We should mention that the equation sets are conservative only for the inviscid case; therefore, in order to conserve mass, we do not apply stabilization to the continuity equation.

3 Spatial discretization of the governing equations

Spatial discretization for the EBG methods, namely continuous Galerkin and discontinuous Galerkin, is conducted by decomposing the domain $\Omega_e \subset \mathbb{R}^3$ into N_e non-overlapping hexahedra elements Ω_e

$$\Omega_e = \bigcup_{e=1}^{N_e} \Omega_e$$

A key property of hexahedral elements is that they allow the use of a tensor-product approach thereby decreasing the complexity (in 3D) from $\mathcal{O}(N^6)$ to $\mathcal{O}(N^4)$ where N is the degree of the polynomial basis. In addition, if we are willing to accept inexact integration of the mass matrix then we can co-locate the interpolation and integration points to simplify the resulting algorithm, in addition to increasing its efficiency without sacrificing too much accuracy (Giraldo, 1998).

Within each element Ω_e are defined basis functions $\psi_j(\mathbf{x})$ to form a finite-dimensional approximation $\mathbf{q}_N$ of $\mathbf{q}(\mathbf{x}, t)$ by the expansion

$$\mathbf{q}_N{}^{(e)}(\mathbf{x}, t) = \sum_{j=1}^{M} \psi_j(\mathbf{x}) \mathbf{q}_j^{(e)}(t)$$

where M is the number of nodes in an element. The superscript (e) indicates a local solution as opposed to a global solution. From here on, the superscript is dropped from our notations since we are solely interested in EBG methods.

The 3D basis functions are formed from a tensor product of the 1D basis functions in each direction as

$$\psi_{ijk}(\xi, \eta, \zeta) = \psi_i(\xi) \otimes \psi_j(\eta) \otimes \psi_k(\zeta)$$

where the 1D Lagrange basis functions are defined on [–1, 1] as

$$\psi_i(\xi) = \prod_{\substack{j=1 \\ j \neq i}}^{N+1} \frac{\xi - \xi_j}{\xi_i - \xi_j}$$

where $\{\xi_i\}_1^M$ is the set of interpolation points in $[-1, 1]$. In a nodal Galerkin approach, $\psi_i(\xi)$ are Lagrange polynomials associated with a specific set of points; here we choose the LGL points $\{\xi_i\} \in [-1, 1]$ which are the roots of

$$(1 - \xi^2)\mathcal{P}_{N'}(\xi)$$

where $\mathcal{P}_N(\xi)$ is the Nth degree Legendre polynomial. These points are also used for integration with quadrature weights given by

$$\omega_i = \frac{2}{N(N+1)} \left(\frac{1}{\mathcal{P}_N(\xi_i)} \right)^2$$

This choice of Lagrange functions gives the Kronecker delta property

$$\psi_i(\xi_j) = \delta_{ij}$$

which, for the 3D basis functions, yields

$$\psi_{ijk}(\xi_a, \eta_b, \zeta_c) = \delta_{ai} \otimes \delta_{bj} \otimes \delta_{ck}$$

Unfortunately, the Kronecker delta property does not hold for the derivatives of the basis functions. However, in the case of tensor-product elements, there exists a simplification that will tremendously decrease the cost of evaluation of derivatives and also the associated storage space in case they are stored as matrix coefficients. Let us write the derivatives in the following way

$$\begin{aligned}
\frac{\partial \psi_{ijk}}{\partial \xi}(\xi_a, \eta_b, \zeta_c) &= \frac{\mathrm{d}\psi_i}{\mathrm{d}\xi}(\xi_a) \otimes \delta_{bj} \otimes \delta_{ck} \\
\frac{\partial \psi_{ijk}}{\partial \eta}(\xi_a, \eta_b, \zeta_c) &= \delta_{ai} \otimes \frac{\mathrm{d}\psi_j}{\mathrm{d}\eta}(\eta_b) \otimes \delta_{ck} \\
\frac{\partial \psi_{ijk}}{\partial \zeta}(\xi_a, \eta_b, \zeta_c) &= \delta_{bj} \otimes \delta_{ai} \otimes \frac{\mathrm{d}\psi_k}{\mathrm{d}\zeta}(\zeta_c)
\end{aligned} \tag{6}$$

Therefore, for tensor-product elements, we need to consider only $3N$ nodes instead of N^3 when computing derivatives at a given node. If matrices are built to solve the system of equations, the storage requirement would increase in proportion to the polynomial order $\mathcal{O}(N)$ instead of $\mathcal{O}(N^3)$. This saving is due to the fact that we only have to compute and store $\frac{\mathrm{d}\psi}{\mathrm{d}\chi}(\chi)$ where χ is one of the following: ξ, η, ζ. The derivatives with respect to the physical coordinates $\mathbf{x} = (x, y, z)$ are computed using the Jacobian matrix transformation

$$\nabla \phi = J \hat{\nabla} \phi$$

where $\hat{\nabla}$ is the derivative with respect to the reference coordinates (ξ, η, ζ) and

$$J = \begin{bmatrix}
\frac{\partial \xi}{\partial x} & \frac{\partial \xi}{\partial y} & \frac{\partial \xi}{\partial z} \\
\frac{\partial \eta}{\partial x} & \frac{\partial \eta}{\partial y} & \frac{\partial \eta}{\partial z} \\
\frac{\partial \zeta}{\partial x} & \frac{\partial \zeta}{\partial y} & \frac{\partial \zeta}{\partial z}
\end{bmatrix} \tag{7}$$

3.1 Continuous Galerkin method

Starting from the differential form of the Euler equations in vector notation, shown in equation (4), and then expanding with basis functions, multiplying by a test function ψ_i, and integrating yields the element-wise formulation

$$\int_{\Omega_e} \psi_i \frac{\partial \mathbf{q}_N}{\partial t} \mathrm{d}\Omega_e + \int_{\Omega_e} \psi_i \nabla \cdot \mathcal{F} \mathrm{d}\Omega_e = \int_{\Omega_e} \psi_i \mathcal{S}(\mathbf{q}_N) \mathrm{d}\Omega_e \tag{8}$$

Integrating the second term by parts ($\psi_i \nabla \cdot \mathcal{F} = \nabla \cdot (\psi_i \mathcal{F}) - \nabla \psi_i \cdot \mathcal{F}$) yields

$$\int_{\Omega_e} \psi_i \frac{\partial \mathbf{q}_N}{\partial t} \mathrm{d}\Omega_e + \int_{\Gamma_e} \psi_i \hat{\mathbf{n}} \cdot \mathcal{F} \mathrm{d}\Gamma_e - \int_{\Omega_e} \nabla \psi_i \cdot \mathcal{F} \mathrm{d}\Omega_e = \int_{\Omega_e} \psi_i \mathcal{S}(\mathbf{q}_N) \mathrm{d}\Omega_e \tag{9}$$

where $\hat{\mathbf{n}}$ is the outward pointing normal on the boundary of the element Γ_e. The second term needs to be evaluated only at physical boundaries because the fluxes to the left and right of element interfaces are always equal at interior boundaries, i.e. $\mathcal{F}^+ = \mathcal{F}^-$. Equations (8) and (9) are the *strong* and *weak* CG formulations, respectively, with the finite dimensional space defined as a subset of the Sobolev space

$$\mathcal{V}_N^{\mathrm{CG}} = \{\psi \in H^1(\Omega_e) | \psi \in \mathcal{P}_N\}$$

where $\mathcal{P}_N$ defines the set of all Nth degree polynomials. Automatically, $\mathcal{V}_N^{\mathrm{CG}} \in C^0(\Omega_e)$, thus CG solutions satisfy C^0 continuity.

3.2 Discontinuous Galerkin method

For DG, the finite dimensional space is defined as a subset of the Hilbert space that allows for discontinuities of solutions

$$\mathcal{V}_N^{\mathrm{DG}} = \{\psi \in L^2(\Omega_e) | \psi \in \mathcal{P}_N\}$$

Therefore $\mathcal{F}^+$ and $\mathcal{F}^-$ are not equal anymore, hence, we define a numerical flux $\mathcal{F}^*$ as an approximate solution to a Riemann problem to be used in the weak-form DG

$$\int_{\Omega_e} \psi_i \frac{\partial \mathbf{q}_N}{\partial t} \mathrm{d}\Omega_e + \int_{\Gamma_e} \psi_i \hat{\mathbf{n}} \cdot \mathcal{F}^* \mathrm{d}\Gamma_e - \int_{\Omega_e} \nabla \psi_i \cdot \mathcal{F} \mathrm{d}\Omega_e = \int_{\Omega_e} \psi_i \mathcal{S}(\mathbf{q}_N) \mathrm{d}\Omega_e \tag{10}$$

where the Rusanov flux, suitable for hyperbolic equations, is defined as

$$\mathcal{F}(\mathbf{q})^* = \{\mathcal{F}(\mathbf{q})\} - \hat{\mathbf{n}} \frac{|\hat{\lambda}|}{2} [\![\mathbf{q}]\!]$$

where $|\hat{\lambda}|$ is the speed of sound, $\{\}$ represent an average and $[\![\,]\!]$ represent a jump across a face (from Ω_e to its neighbor). If C^0 continuity is enforced on the weak-form DG in equation(10), i.e. $\mathcal{F} = \mathcal{F}^*$, it reduces to the weak-form CG in equation (9).

A strong-form DG that resembles equation (8) more, can be obtained by applying a second integration by parts on the flux integral to remove the smoothness constraint on the test function ψ_i as follows

$$\int_{\Omega_e} \psi_i \frac{\partial \mathbf{q}_N}{\partial t} \mathrm{d}\Omega_e + \int_{\Gamma_e} \psi_i \hat{\mathbf{n}} \cdot (\mathcal{F}^* - \mathcal{F})(\mathbf{q}_N) \mathrm{d}\Gamma_e + \int_{\Omega_e} \psi_i \nabla \cdot \mathcal{F}(\mathbf{q}_N) \mathrm{d}\Omega_e = \int_{\Omega_e} \psi_i \mathcal{S}(\mathbf{q}_N) \mathrm{d}\Omega_e \tag{11}$$

Again, if C^0 continuity is enforced on the strong-form DG formulation, i.e. $\mathcal{F} = \mathcal{F}^*$ at interior edges, it simplifies to the strong-form CG formulation in equation (8) (see Abdi and Giraldo (2016) for details).

3.3 Unified CG and DG

The element-wise matrices for both CG and DG are assembled to form global matrices via an operation commonly known as global assembly or direct stiffness summation (DSS). Even though the local matrices are the same for both methods, the DSS operation yields different global matrices. CG is often implemented through a global grid point storage scheme where elements share LGL nodes at faces so that C^0 continuity is satisfied automatically. Therefore, the DSS operation for CG accumulates values at shared nodes, while that for DG simply puts the local element matrices in their proper location in the global matrix. DG uses a local element-wise storage scheme because discontinuities (jumps) at element interfaces are allowed. The standard implementation of CG and DG often follows these two different approaches of storing data; however, CG can be recast to use local element-wise storage as well. To do so, we must explicitly enforce equality of values on the right and left of element interfaces by accumulating and then distributing back (gather–scatter) values at shared nodes for both the mass matrix and right-hand-side (RHS) vector. The gather–scatter operation is the coupling mechanism for CG, without which the problem is under-specified. DG achieves the same via the definition of the numerical flux $\mathcal{F}^*$ at element interfaces, which is used by both elements sharing the face. A detailed explanation of the unified CG and DG implementation of NUMA can be found in Abdi and Giraldo (2016).

4. Temporal discretization of the governing equations

The time integrator used is a low-storage explicit Runge–Kutta (LSERK) method proposed in Carpenter and Kennedy (1994). It is a five-stage fourth-order RK method that requires only two storage locations, which is half of that required by the conventional high-storage fourth-order RK method. The added cost due to one more stage evaluation is offset by the larger stable timestep Δt the method allows. Each successive stage is written on to the same register without erasing the previous value. We need to store previous values of the field variable $\mathbf{q}$ and its residual $\mathrm{d}\mathbf{q}$ of size N each, thereby, resulting in a $2N$-storage scheme. Given the initial value problem

$$\frac{\mathrm{d}\mathbf{q}}{\mathrm{d}t} = \mathcal{R}(\mathbf{q}) \text{ with } \mathbf{q}(t_0) = \mathbf{q}_0$$

the updates at each stage j are conducted as follows

$$\mathrm{d}\mathbf{q}_j = A_j\,\mathrm{d}\mathbf{q}_{j-1} + \Delta t\mathcal{R}(\mathbf{q}_{j-1})$$
$$\mathbf{q}_j = \mathbf{q}_{j-1} + B_j\,\mathrm{d}\mathbf{q}_j$$

where A_j and B_j are constant coefficients for each stage given in Table 2

Explicit RK methods have a stringent Courant–Friedrichs–Lewy (CFL) requirement that often prohibits them from being used in operational settings. NUMA includes Implicit–Explicit (IMEX) methods that allow for much larger timesteps, however, those have not yet been ported to the GPU. The first goal of the GPU project focuses on porting explicit time integration methods, which are known to scale well on many processors and are also easier to port to GPUs. Implicit methods require the solution of a coupled system of linear equations; therefore, depending on the chosen iterative solver and pre-conditioner, performance on a cluster of computers and GPUs may be severely impacted. For this reason, we reserve the porting of the implicit solvers in NUMA to a future study.

5 Porting NUMA to the GPU

This section describes the implementation of the unified CG and DG NUMA on the GPU using the OCCA programming language (Medina ET AL., 2014). Before we delve into details of the implementation, a few words on GPU computing in general and design considerations are warranted. GPUs provide the most cost-effective computing power to date, however, they come with a challenge of adapting existing code originally written for the CPU to a GPU platform.

Table 2 Coefficients of the five-stage LSERK time integrator.

Stage	A	B
1	0	0.097618354
2	0.481231743	0.412253292
3	−1.049562606	0.440216964
4	−1.602529574	1.426311463
5	−1.778267193	0.197876053

5.1 Challenges

First of all, the candidate program to be ported to the GPU should be able to handle massively fine grained parallelism via threads. Even though current general purpose GPU computing offers a lot more flexibility than the days when they were exclusively used for image rendering, there are still limitations on what can be done efficiently on GPUs. Single Instruction Multiple Data (SIMD) programs suited for vector machines are automatically candidates for porting to GPUs. More flexibility is achieved on the GPU by limiting SIMD computation to a small group of threads, 32 threads known as a *warp* in NVIDIA terminology, and then scheduling multiple warps to work on different tasks. In the code design phase, it is often convenient to think of warps as the smallest computing unit for the following reason. If even one thread in a warp decides to do a different operation, warp divergence occurs in which all threads in a warp have to do operations twice resulting in a 50% performance loss.

The second issue concerns memory management. Though the many cores in GPUs provide a lot of computational power, they can only be harnessed fully if unrestricted by memory bandwidth limitations. Programs running on a single-core CPU are often compute-bound because more emphasis is given to data caching in CPU design. In contrast, most of the chip area in GPUs is devoted to compute units, and as a result, programs running on a GPU tend to be memory-bound. Programmers have to carefully manage the different memory resources available in GPUs. To give an idea of the complexity of memory management, we briefly describe the six types of memory in NVIDIA GPUs: global, local, texture, constant, shared and register memory ordered in highest to lowest latency. Register memory is the fastest but is limited in size and only visible to one thread. Shared memory is fast and visible to a *block*, a group of warps, and therefore it is an invaluable means of communication between threads. Constant and texture memory are read-only memory that can be used to reduce memory traffic. Local memory is cached but is only accessible by one thread; automatic variables that cannot be held in registers are offloaded to the slow local memory. Global memory, which is accessible by all threads, is the main memory of GPUs where the data is stored.

5.2 Design choices

Global memory bandwidth limitation and high latency of access is often the bottleneck of performance in GPU computing. To minimize its impact on performance, memory transactions can be coalesced for a group of threads accessing the same block in memory. The warp scheduler also helps to alleviate this problem by swapping out warps that are waiting for a global memory transaction to complete for those that are ready to go. There are two approaches to storing data. The first approach, Array of Structures (AoS), stores all variables at a given LGL node contiguously in memory. This is suitable if computation is done for all the variables in one pass. If, on the other hand, a subset of the variables are required at a time, a second approach, Structure of Arrays (SoA), is suitable. While the SoA often degrades performance on the CPU due to reduced cache efficiency, it can significantly improve performance on the GPU because of coalesced memory transactions for a warp. The approach we use is a mix of these two methods similar to the AoSoFA (Array of Structures of Fixed Arrays) described in Allard et al. (2011), in which data for each element is stored in an SoA manner, and thus an AoS for the whole domain. Using this approach, scalar data for all nodes in an element is stored contiguously in memory; this is repeated similarly for each scalar variable. Variables that are often accessed together, for instance coordinates (x,y,z) or velocity (u,v,w) can be stored as one *float3* on the GPU. A pseudo-code for the data layout in CPU and GPU modes is shown in Algorithm 1.

Our choice of data layout is influenced by our design decision to do computation on an element-by-element basis, for instance launching as many threads as the number of nodes for computing volume integrals, and as many as face nodes for surface integrals (see sections 5.3.1 and 5.3.2). We should note here that our approach has a downside in that the number of threads launched for processing an element could be small with low-order polynomial approximations; also the number of threads may not be a multiple of the warp size. We provide solutions to this problem by processing multiple elements per block as will be explained in the coming sections. In the SoA approach, these two problems do not exist and the appropriate number of threads that fit the GPU device could be launched to process LGL nodes, even from different elements simultaneously. The SoA approach may be better for porting code to the GPU using, for instance, OpenACC or other pragma-based programming languages where the user has less control of the device.

Algorithm 1 Pseudo-code for data layout.

```
#ifdef OPENMP        ▷ CPU mode: Define an AoS for each node where Jx,Jy,Jz are stored contiguously.
occaPrivateArray(float4, data, 3)
#define Jx data[0]
#define Jy data[1]
#define Jz data[2]
#else                ▷ GPU mode: Define AoSoFA where Jx is contiguously stored for all (N^3) nodes in each element.
occaPrivate(float4, Jx);
occaPrivate(float4, Jy);
occaPrivate(float4, Jz);
#endif
```

5.3 Unified CG and DG on the GPU

The implementation of CG done within the DG framework differs only by the final DSS step required for imposing the C^0 continuity constraint instead of using the numerical flux. Therefore, first we explain the implementation details of nodal DG on the GPU and then that of the DSS operation later. The three major computations in DG are implemented in separate OCCA kernels: volume integration, surface integration and timestep update kernels. Other major kernels are the boundary kernel required for imposing boundary conditions, the project kernel for applying the DSS operation for CG, and two kernels for stabilization: a Laplacian diffusion kernel for applying second-order artificial viscosity to be used with CG, and a kernel for computing the gradient required by the Local Discontinuous Galerkin (LDG) method used for stabilizing DG; in future work, we will select one stabilization method/kernel for both methods using the primal form of the elliptic problem. For the strong-form DG discretization of the Euler equations, the kernels represent the following integrals

$$\underbrace{\int_{\Omega_e} \psi \frac{\partial \mathbf{q}}{\partial t} \mathrm{d}\Omega_e}_{\text{Update kernel}} + \underbrace{\int_{\Omega_e} \psi(\nabla \cdot \mathcal{F} - \mathcal{S}) \mathrm{d}\Omega_e}_{\text{Volume kernel}} + \underbrace{\int_{\Gamma_e} \psi \hat{\mathbf{n}} \cdot (\mathcal{F}^* - \mathcal{F}) \mathrm{d}\Gamma_e}_{\text{Surface kernel}} = \underbrace{\int_{\Omega_e} \psi(\nabla \cdot \mu \nabla \mathbf{q}) \mathrm{d}\Omega_e}_{\text{Diffusion kernel}} \tag{12}$$

5.3.1 Volume kernel. The volume and surface integration kernels are written in such a way that a CUDA thread block processes one or more elements, and a thread processes contributions from a single LGL node, i.e. the one-node-per-thread approach we mentioned in the Introduction. Gandham et al. (2014) mention that for low-order polynomial approximations, performance can be improved by as much as $5\times$ by processing more than one element per block. This is especially true for 2D elements that were used in their study, which have fewer nodes than the 3D elements we are using in this work. The reason for this variation in performance with the number of elements processed per block is the

Algorithm 2 GPU algorithms for computing gradient, divergence and Laplacian.

```
Procedure GradDiv (q,grad,div,compute)                    ▷ Compute gradient or divergence
  Memory fence
  for k, j, i ∈ {0 ... Nq}} do                            ▷Load field variables into shared memory
    sq[k][j][i] = q
  Memory fence
  for {k, j, i ∈ {0 ... Nq} do
    qx = 0; qy = 0; qz = 0;
    for n ∈ {0 ... Nq} do                                 ▷Compute local gradients
      qx + = sD[i][n] × sq[k][j][n]                       ▷ sD are ∇Ψ at LGL nodes pre-loaded to shared memory.
      qy + = sD[j][n] × sq[k][n][i]
      qz + = sD[k][n] × sq[n][j][i]
    if compute = GRAD then
      grad·x = (qx × Jrx + qy × Jsx + qz × Jtx)           ▷ Js are coefficients of the Jacobian matrix J
      grad·y = (qx × Jry + qy × Jsy + qz × Jty)
      grad·z = (qx × Jrz + qy × Jsz + qz × Jtz)
else if compute = DIVX then
      div = (qx × Jrx + qy × Jsx + qz × Jtx)
else if compute = DIVY then
      div + = (qx × Jry + qy × Jsy + qz × Jty)
else if compute = DIVZ then
      div + = (qx × Jrz + qy × Jsz + qz × Jtz)

Procedure GRAD (q,grad)                                   ▷ Compute gradient of a scalar field
  call GRADDIV (q,grad,-,GRAD)

Procedure DIV (q,div)                                     ▷ Compute divergence of a vector field
  call GRAD DIV (q· x,-,div,DIVX)
  call GRAD DIV (q· y,-,div,DIVY)
  call GRAD DIV (q· z,-,div,DIVZ)

Procedure LAP(q, lap)                                     ▷ Compute Laplacian of a scalar field
  call GRAD (q,gq)
  call DIV (gq,lap)
```

need for a block size that best fits the underlying hardware limits. In traditional GPU kernels, for instance the timestep update kernel discussed in section 5.3.3, thread blocks are sized as multiples of the warp size (32 threads) for best performance. However, for the volume integration kernels, our algorithms are designed such that one thread processes one LGL node, therefore the number of threads launched is not a multiple of the warp size but the number of nodes.

The main operation in the volume kernel is computing gradients of the following eight variables (shown in Algorithm 2): five prognostic variables (ρ, U, V, W, Θ), pressure P and two variables for moisture (here, we omit precipitation). The gradient of four variables, which are stored as one *float4*, can be computed together for efficiency. The current work does not include support for tracer transport, nor do we employ the moisture dynamics even though the gradient is computed. Once the gradients are calculated, we can construct the divergence and complete the contribution of the volume integration to the RHS vector as shown in Algorithm 3.

For low-order polynomials, we can launch one thread per node and perhaps more by processing multiple elements per block. This approach works for a maximum polynomial order of seven on older GPU cards and up to order eight on the Fermi architecture. The reason why we cannot use this approach for higher order polynomials than seven is two fold: first, the number of threads in a block $((7+1)^3 = 512)$ approaches the hardware block size limit. Second, we also approach the shared memory limit at around this polynomial order. Therefore, we use two different approaches for volume integration for polynomial orders less than seven (low order) and greater than seven (high order). For low-order polynomials, we can pre-load all the element data (the two *float4s* to shared memory at startup, and then never read from global memory again until the kernel completes).

We can overcome the thread block size limitation for high-order polynomial approximation by launching only the required number of threads to process one slice of a 3D element, i.e. N_{LGL}^2 nodes, as shown in Figure 1. Then, we consider three ways of exploiting the shared memory shown in pseudo-code in Algorithm 4. The first approach, which we call the *naïve approach*, does not use shared memory but relies solely on the L1 cache if available. Otherwise, data is read directly from global memory every time it is required. We can optimize this approach by adjusting the hardware division of L1 cache to shared memory to be 48 kb/16 kb instead of the default 16 kb/48 kb in the K20X GPU. Ignoring

Algorithm 3 Outline of a combined volume kernel for processing N_k elements per block with N_s slice workers. There are N_q quadrature points, slices per element for volume kernels and N_f faces, for surface kernels.

Procedure VOLUMEKERNEL (q,R)
  Shared data[N_k][N_q][N_q][N_q] ▷ Extended shared memory array
  for outerId0 **do**
    for innerId2 **do**
      wId = innerId2 **mod** N_s ▷ Slice worker Id
      eId = innerId2 **div** N_s ▷ Multiple element processing
      for sId = wId **to** N_q **step** N_s **do** ▷ N_q slices to work on
        e = $N_k \times$ outerId0 + eId ▷ Element id

        call GRAD($qa, \nabla qa$) ▷ Compute gradient of (U,V,W,p) as one *float4* variable *qa*
        $DU = \nabla_x U + \nabla_y V + \nabla_z W$
        $R(\rho) = DU$
        $R(\Theta) = \theta \times DU$
        $R(U) = U \times DU + \nabla_x p + \nabla U \cdot \boldsymbol{U}$
        $R(V) = V \times DU + \nabla_y p + \nabla V \cdot \boldsymbol{U}$
        $R(W) = W \times DU + \nabla_y p + \nabla W \cdot \boldsymbol{U}$

        call GRAD($qb, \nabla qb$) ▷ Compute gradient of $(\rho, \Theta, -, -)$ as one *float4* variable *qb*
        $DR = \boldsymbol{U} \cdot \nabla \rho$
        $R(\Theta) = \Theta \times DR - \boldsymbol{U} \cdot \nabla \Theta$
        $R(u) = U \times DR$
        $R(v) = V \times DR$
        $R(w) = W \times DR$

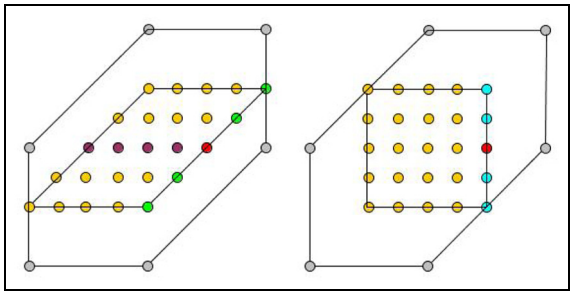

Figure 1. Volume integral contribution of a horizontal and vertical slice of a 3D element with 4th degree polynomial approximation. Due to the use of the tensor-product approach for hexahedral elements, contributions to a given node (red dot) come only from those collinear with it along the *x*-, *y*- and *z*-directions. The horizontal volume kernel computes the contribution from the purple and green nodes on the horizontal slice, and the vertical volume kernel adds the contribution from the light-blue nodes on the vertical slice.

Algorithm 4 Methods for exploiting shared memory.

Global Q[NELEM[Nq][Nq][Nq] ▷ Q is in global memory
Shared sQ[N][N] ▷ Pre-loaded slice data in shared mem
Private pQ[N] ▷ Column data loaded to private mem (registers) with each of the N^2 threads.
Pre-load sQ and pQ
if naïve **then**
  dataH = Q[e][sl][j][i]
  dataV = Q[e][k][j][i]
else if SHARED-1 **then**
  dataH = sQ[j][i]
  dataV = Q[e][k][j][i]
else if SHARED-2 then
  dataH = sQ[j][i]
  dataV = pQ[k]

cache effects, the naïve approach reads $3N_{\mathrm{LGL}}$ values from memory to compute the gradient of a variable at a node, for a total of $N_{\mathrm{LGL}}^3 \times 3N_{\mathrm{LGL}}$ memory reads. The second approach, henceforth called *Shared-1* loads a slice of data to shared memory, then computes the contribution to the gradient from those nodes on the slice. The data on the slice is re-used between the N_{LGL}^2 nodes on the same plane, therefore, a total of $N_{\mathrm{LGL}}^3 \times N_{\mathrm{LGL}}$ memory reads are required. The third approach, henceforth called *Shared-2*, extends the previous method by storing the column of data in register as suggested in Micikevicius (2009). The column of data may not fit in registers in which case it is spilled to CUDA private memory, which is global memory. In the latter case, the method will be the same as the Shared-1 approach with the additional cost of copying data from global-to-global memory. The best case scenario is when N_{LGL}^3 memory reads are required, but this cannot be achieved in practice due to the limited number of registers per thread. The fourth approach does two passes on the data in which the first pass calculates contributions to the gradient from nodes on the same slice, say the $x-y$ plane; the second pass completes the gradient calculation by loading $x-z$ slices, and adding the contributions from nodes in the z-direction. This approach always requires $N_{\mathrm{LGL}}^3 \times 2$ memory reads, and its implementation requires separate horizontal and vertical volume kernels that exploit shared memory within the respective planes.

Algorithm 5 Surface kernel.

```
map[3][2] = ((0,5),(1,3),(2,4))                    ▷ Pairs of faces, shown in Figure 2, for parallel computation
Procedure SURFACEKERNEL (q,R)
  for outerId0 do
    for innerId2 do
      wId = innerId2 mod N_s                       ▷ Slice worker Id
      eId = innerId2 div N_s                       ▷ Element Id
      for wId to 2 step N_s do
        for b = 0 to 2 do
          sId = map[b][wId];                       ▷ Get face
          for j,i ∈ {0 ... Nq}do
            e = N_k×outerId0 + eId
            Load face normal n̂ and lift coefficient ℒ   ▷ ℒ = (w_ij J_ij)/(w_ijk J_ijk)
            Load q⁺ and q⁻ for current node and adjoining node in the other element
            Compute maximum wave speed |λ| = |n̂×u| + √(γp/ρ)
            Compute Rusanov flux ℱ(q)* = {ℱ(q)} − n̂ (|λ|/2) [[q]]
            R +  = ℒ×n̂ · (ℱ(q)* − ℱ(q))
```

Even though the slicing approach helps to handle higher order polynomial approximations, it hurts performance on the other end of the spectrum. Assuming 512 threads per block and a hardware limit of 8 blocks per multi-processor, a 2D kernel using 3rd degree polynomial approximations will require $8 \times (3+1)^2 = 128$ threads, which yields 25% efficiency; on the other hand a 3D kernel will occupy 100% of the device because $8 \times (3+1)^3 = 512$ threads are launched per multi-processor. We would like to run with high-order polynomial approximations and also have kernels that are efficient for low-order polynomial approximations. We should mention here that occupancy is not the only indicator of performance, and one could obtain better results with fewer threads and better instruction level parallelism (Volkov, 2010).

These two competing goals of optimizing kernels for high-order and low-order polynomials can be handled separately with different kernels optimized for each. More convenient is to write the volume kernel in such a way that it can process multiple elements in a thread block with one or more slice workers simultaneously. For this reason, the volume, surface and gradient kernels accept parameters N_k, for number of elements to process per block, and N_s, for the number of *slice workers* per element. We should note here that due to the runtime compilation feature of OCCA, parameters such as the polynomial order are constants, as a result, kernels are optimized for the selected set of parameters. For example, with $N_k = 1$ and $N_s = 1$, the kernels produced will be exactly the same as those we had before adding the multiple element per block and slicing approaches. If a kernel uses shared memory to store data for each element processed per block and slice worker, its shared memory consumption will increase in proportion with $N_k \times N_s$, as shown in Algorithm 3.

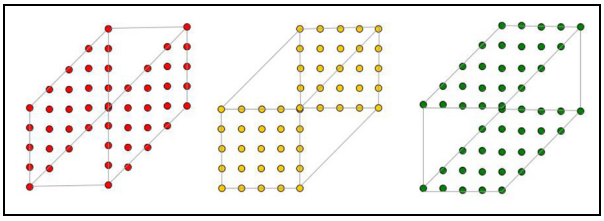

Figure 2. Coloring of faces for parallel computation of the surface integral. Opposing faces can be processed simultaneously because there are no shared edges between them.

5.3.2 Surface kernel. The surface integration, shown in Algorithm 5, is conducted in two stages in accordance with Klöckner et al. (2009): the flux gather stage collects contributions of elements to the numerical flux at face nodes, and the lifting stage integrates the face values back into the volume vector. Lifting, in our case, is a simple multiplication by a factor computed from the ratio of weighted face and volume Jacobians; this is a result of the tensor-product approach in conjunction with the choice of integration rule that results in a diagonal lifting matrix. If the numerical flux at a physical boundary is pre-determined, for instance in the case of a no-flux boundary condition, it is directly set to the prescribed value before lifting. The workload in surface integration can be split into slices similar to that used for volume integration. The number of slices available for parallelization in this case is the number of faces of an element, six for hexahedra. However, since two faces that are adjacent to each other share an edge, they cannot be processed by two slice workers simultaneously. One solution is to reduce the parallelization to pairs of opposing faces, thereby avoiding the conflict that arises at the edges when updating flux terms, as shown in Figure 2. A second option is to use hardware atomic

operations to update the flux terms. However, hardware support for atomic operations on double-precision (DP) floating point operations is not universally supported by all GPUs at this time.

5.2.3 Update kernel. The timestep update kernel is relatively straightforward to implement because we are using explicit time integration, in which new values at a node are calculated solely from old values at the same node. However, explicit timestepping is only conditionally stable depending on the Courant number. The implementation of implicit–explicit and fully implicit timestepping methods, which require the solution of a linear system of equations, is postponed to the future. For now, we implement the low-storage fourth-order RK method of Carpenter and Kennedy (1994) by storing the solution at the previous timestep and its residual. Since there is no distinction between nodes in different elements for this particular kernel, we can select the appropriate block size that best fits the hardware, e.g. 256 in OpenCL.

5.3.4 Project kernel. The DSS operation is implemented in two steps, namely *gather* and *scatter* stages. The DSS kernel, shown in Algorithm 6, accepts a vector of node numbers in Compressed Sparse Row (CSR) format. This vector is used to gather local node values to then put the result in global nodes — which may be mapped into multiple local nodes. One thread is launched for each global node to accumulate the values from all local nodes sharing this global node. As a result, no conflict will arise while accumulating values, because the gather at a node is done sequentially by the same thread. For the single-GPU implementation, we can immediately start the scatter operation, which does the opposite operation of scattering the gathered value back to the local nodes. However, a multi-GPU implementation requires communication of gathered values between GPUs before scattering, as will be discussed in section 6.

Algorithm 6 DSS kernel.

```
Procedure DSSKernel (Q,Qcont,starts,indices,nGlobal,wgt)
  for outerId0 do
    n = outerId0                                   ▷ Global node id
    if n ⩽ nGlobal then
      start = starts[n]      ▷ Read indices of local nodes for the
      DSS operation
      end = starts[n + 1]
      gQ = 0                                  ▷ Gather stage of DSS
      for m = start to end do
        ind = indices[m]                           ▷ Local node index
        if ind ≥ 0 then
          pw = wgt[ind];    ▷ DSS weight computed based on
          lumped mass coefficients
          gQ + = Q[ind] × pw
      Qcont[n] = gQ
      for m = start to end do              ▷ Scatter stage of DSS
        ind = indices[m]
        if ind ≥ 0 then
          Q[ind] = Qcont[n]
```

5.3.5 Diffusion kernels. For the purposes of the current work, we shall use constant second-order artificial viscosity to stabilize both the CG and DG methods in NUMA[2] The stabilizing term, shown in equation (5), is in divergence form $\nabla \cdot (\mu \nabla \mathbf{q})$ so that we will be able to use dynamic viscosity methods in the future. However, we use constant viscosity in the current work, which reduces the stabilizing term to a Laplacian operator $\mu \nabla^2 \mathbf{q}$.

For stabilizing CG, we use the primal form discretization of the Laplacian operator. Let us start with the DG discretization with numerical flux $\mathbf{q}^*$ given in weak form as

$$\int_{\Omega_e} \psi_i \nabla \cdot (\mu \nabla \mathbf{q}) \mathrm{d}\Omega_e = \underbrace{\int_{\Gamma_e} \psi_i \hat{\mathbf{n}} \cdot (\mu \nabla \mathbf{q}^*)}_{\text{surface}} - \underbrace{\int_{\Omega_e} \nabla \psi_i \cdot (\mu \nabla \mathbf{q}) \mathrm{d}\Omega_e}_{\text{volume}} \tag{13}$$

and in the strong form as

$$\int_{\Omega_e} \psi_i \nabla \cdot (\mu \nabla \mathbf{q}) \mathrm{d}\Omega_e = \underbrace{\int_{\Gamma_e} \psi_i \hat{\mathbf{n}} \cdot (\mu \nabla \mathbf{q}^* - \mu \nabla \mathbf{q})}_{\text{surface}} + \underbrace{\int_{\Omega_e} \psi_i \nabla \cdot (\mu \nabla \mathbf{q}) \mathrm{d}\Omega_e}_{\text{volume}} \tag{14}$$

If we, then, ensure C^1 continuity in the CG discretization, i.e. by applying DSS on the gradient so that $\nabla \mathbf{q} = \nabla \mathbf{q}^*$, the surface integral term disappears from the strong-form formulation. The weak-form CG formulation will still retain the surface integral term despite DSS, however, this term needs to be evaluated only at physical boundaries because it cancels at interior boundaries due to $\nabla \mathbf{q}^+ = \nabla \mathbf{q}^-$. In addition, the term completely disappears if *no-flux* boundary conditions are used; dropping the surface integral term in other cases results in an inconsistent method, but something that could still be feasible for the purpose of numerical stabilization. The kernel for computing the volume contribution of the strong-form discretization is already given as the Laplacian procedure in Algorithm 2. The volume kernel for the the weak-form discretization is shown in Algorithm 7. The first step in this kernel is to load the field variable $\mathbf{q}$ into the fast shared memory. Then, we compute and store the local gradients at each LGL node similar to what is done in

Algorithm 7 Laplacian diffusion kernel.

```
Procedure Laplace Q,rhs,nu
  Shared sq,sqr,sqs,sqt all arrays of size of [Nq][Nq][Nq]
  Memory fence
  for {k, j, i ∈ {0 ... Nq} do                    ▷ Load field variables into shared memory}
    sq[k][j][i] = q
  Memory fence
  for k, j, i ∈ {0 ... Nq} do
    qr = 0; qs = 0; qt = 0;                       ▷ Compute local gradients in r-s-t
    for n ∈ {0 ... Nq} do
      qr + = sD[i][n] × sq[k][j][n];              ▷ sD are ∇ψ at LGL nodes pre-loaded to shared memory.
      qs + = sD[j][n] × sq[k][n][i];
      qt + = sD[k][n] × sq[n][j][i];
    sqr[k][j][i] = μ (G11 × qr + G12 × qs + G13 × qt);   ▷ Gs are coeff. of the symmetric JJ^T matrix
    sqs[k][j][i] = μ (G12 × qr + G22 × qs + G23 × qt);
    sqt[k][j][i] = μ (G13 × qr + G23 × qs + G33 × qt);
Memory fence
for k, j, i ∈ {0 ... Nq} do
    lapq = 0
    for n ∈ {0 ... Nq} do
      lapq + = sD[n][i] × sqr[k][j][n];
      lapq + = sD[n][j] × sqs[k][n][i];
      lapq + = sD[n][k] × sqt[n][j][i];
    rhs - = Jinv × lapq
```

the volume kernel. The shared memory requirement of this kernel is rather high due to the need for temporarily storing the gradients in addition to the field variables.

On the other hand, the mixed-form stabilization method we use for DG, i.e. by computing and storing the gradient in global memory, puts less stress on shared memory requirement, while being potentially slower. The same kind of optimizations used for the volume kernel, such as splitting into slices and multiple elements per block processing, can be used here as well. After computing the local gradients, the $\nabla\psi_i \cdot \mu\nabla\mathbf{q}_j$ term can be computed immediately afterwards — which is represented by the combined geometric factors JJ^{T}. Note that we use local memory fences to synchronize the read/write operations in shared memory. The fact that we use a discontinuous space even for CG forces us to apply DSS on both $\mathbf{q}$, for which we already applied DSS at the end of the timestep or RK-stage, and $\nabla\mathbf{q}$, for which we ignore DSS for efficiency reasons discussed later in this section. In the case of hyperviscosity of order three or more, the DSS on $\nabla\mathbf{q}$ may be required to ensure at least C^1 continuity.

For stabilizing DG, we use the mixed form of Bassi and Rebay (1997). The viscous term $\nabla \cdot (\mu\nabla\mathbf{q})$ needed for stabilizing the Euler equations in equation (5) requires us to first compute the gradient $\nabla\mathbf{q}$. We can write the computation of the stabilizing term in mixed form as follows

$$\begin{aligned} \nabla\mathbf{q} &= \mathbf{Q} \\ \nabla \cdot (\mu\nabla\mathbf{q}) &= \nabla \cdot (\mu\mathbf{Q}) \end{aligned} \tag{15}$$

where $\mathbf{Q}$ is the auxiliary variable. Because we are evaluating the stabilizing term explicitly, we can solve the equations in a straightforward decoupled manner Bassi and 1997). The strong-form DG discretization of the first part of equation (15) is as follows

$$\int_{\Omega_e} \psi_i \mathbf{q} \mathrm{d}\Omega_e = \underbrace{\int_{\Gamma_e} \psi_i \hat{\mathbf{n}} \cdot (\mathbf{q}^* - \mathbf{q}) \mathrm{d}\Gamma_e}_{\text{surface}} + \underbrace{\int_{\Omega_e} \psi_i \nabla\mathbf{q} \mathrm{d}\Omega_e}_{\text{volume}} \tag{16}$$

We should note that the surface integral term is zero for strong-form CG because $\mathbf{q}^* = \mathbf{q}$ due to continuity. Once we compute $\mathbf{Q}$, we can then compute the viscous term via the discretization

$$\int_{\Omega_e} \psi_i \nabla \cdot (\mu\nabla\mathbf{q}) \mathrm{d}\Omega_e = \underbrace{\int_{\Gamma_e} \psi_i \hat{\mathbf{n}} \cdot (\mu\mathbf{Q}^* - \mu\mathbf{Q}) \mathrm{d}\Gamma_e}_{\text{surface}} + \underbrace{\int_{\Omega_e} \psi_i \nabla \cdot (\mu\mathbf{Q}) \mathrm{d}\Omega_e}_{\text{volume}} \tag{17}$$

According to Bassi and Rebay (1997), we use centered flux17es for both $\mathbf{q}$ and $\mathbf{Q}$ such that $\mathbf{q}^* = \{\mathbf{q}\}$ and $\mathbf{Q}^* = \{\mathbf{Q}\}$. The mixed form is implemented directly by first computing the volume integral of the gradient in equation (16) using Algorithm 2, and then modifying the result with the surface integral contribution computed using centered fluxes $\mathbf{q}* = \{\mathbf{q}\}$. It is necessary to

store **Q** in global memory, unlike the case for CG, and compute the surface integral using a different kernel, because data is required from neighboring elements. This difficulty would have also manifested itself in CG if we chose to gather–scatter **Q**, which would require a separate kernel for similar reasons, and force us to use the mixed form. The fact that we need this term just for stabilization, and not, for instance for the implicit solution of the Poisson problem, gives us some leeway to its implementation on the GPU for performance reasons. However, in the CPU version of NUMA we apply the DSS operator (which requires inter-process communication) right after computing the gradient **Q**. The kernel for computing the surface gradient fluxes is similar to the surface integration kernel discussed in section 5.3.2 — with the only difference being that we use centered fluxes instead of the upwind-biased Rusanov flux. Finally, the volume and surface integral contributions of the viscous term in equation (17) are added to the RHS vector in the volume and surface kernels, respectively. In the future we will study stabilization of DG using the Symmetric Interior Penalty Method (SIPG)– which shares the same volume integration kernel as the weak-form CG stabilization method.

6 Multi-GPU implementation

The ever increasing need for higher resolution in NWP implies that such large-scale simulations cannot be run on a single GPU card due to memory limitations.A practical solution is to cluster cheap legacy GPU cards and break down the problem into smaller pieces that can be handled by a single GPU card; however, this necessitates communication between GPUs, which is often a bottleneck of performance. We extend our single-GPU implementation of NUMA to a multi-GPU version using the existing framework for conducting multi-CPU simulations on distributed memory computers (see Kelly and Giraldo (2012) for details). The communication between GPUs is done indirectly through CPUs,which is the reason why we were able to use the existing MPI infrastructure. We should note that the latest technology in GPU hardware allows for direct communication between GPUs, but the technology is not yet mature and also the GPU cards are more expensive.

6.1 Multi-GPU parallelization of EBG methods

The goal of parallelizing NUMA to distributed memory CPU clusters has already been achieved in Kelly and Giraldo (2012), in which linear scalability of up to 10s of thousands of CPUs was demonstrated. More recently the scalability of the implementation was tested on the Mira supercomputer, located at the Argonne National Laboratory, using 3.1 million MPI ranks (Müller et al., 2016). NUMA achieved linear scalability for both explicit and 1D IMEX time integration schemes in global numerical weather prediction problems. The current work extends the capability of NUMA to multi-GPU clusters, which are known to deliver much more floating point operations per second (FLOPS/s) than multi-CPU clusters. In the following sections, we describe the parallel grid generation and partitioning, multi-GPU CG and DG implementations.

6.1.1 Parallel grid generation. The grid generation and partitioning stages are done on the CPU and then geometric data is copied to the GPU once at startup. The reason for this choice is mainly a lack of robust parallel grid generator software with a capability of Adaptive Mesh Refinement (AMR) on the GPU. Originally NUMA used a local grid generation code and the METIS graph partitioning library for domain decomposition; however, the need for parallel grid generation and parallel visualization output processing was exposed while conducting tests on the Mira supercomputer. Even though a parallel version of METIS (ParMETIS) exists, we chose to adopt the parallel hexahedral grid generation and partitioning software p4est (Burstedde et al., 2011) mainly because of the latter's capability of parallel AMR.In static AMR mode, p4est is in effect a parallel grid generator. Dynamic AMR requires copying geometric data to the GPU more than once, i.e. whenever AMR is conducted. For this reason, recomputing all geometric data on-the-fly on the GPU could potentially improve performance. ParMETIS is a graph partitioning software and as such is not capable of mesh refinements.

6.1.2 Multi-GPU CG. The coupling between subdomains in the CG spatial discretization is achieved by the DSS operator, which imposes C^0 continuity of solutions at element interfaces. The DSS operator is applied both to the mass matrix and the RHS vector. Therefore, a multi-GPU implementation of CG requires communication between GPUs only for applying DSS; in fact, we require GPU kernels for applying DSS only on the RHS vector because the construction of the mass matrix is done on the CPU. However, to apply DSS on the RHS vector, we need several kernels. Algorithm 8

Algorithm 8 DSS on the GPU for the RHS vector.

Procedure DSS *RHS*
  Gather RHS ▷ See Algorithm 6 for details
  Copy boundary data to contiguous block of global memory
  Copy boundary data to CPU
  CPUs communicate and form the global RHS
  CPUs copy the assembled RHS back to the GPU
  for all neighbors **do** ▷ To avoid conflict in RHS update
    Boundary data is used to update the RHS vector
  Scatter RHS ▷ See Algorithm 6 for details

outlines the steps required for applying DSS in a multi-GPU CG implementation. First, we need a kernel to do the intra-GPU gather operation on the RHS vector. Then, the values at inter-GPU boundaries are copied to a contiguous block of GPU global memory, after which the data is copied to the CPU. CPUs, then, communicate the boundary data to construct the global RHS using the existing MPI infrastructure in NUMA. Once the CPUs complete the DSS operation, the CPUs copy the boundary data back to the GPU global memory. Contribution from neighboring processors are processed one-by-one to update the RHS vector; without this 'coloring' of neighboring processors, conflicts in RHS updates can occur at shared edges and corner nodes of elements. The last stage does the intra-GPU scatter operation of DSS.

6.1.3 Multi-GPU DG. The coupling between subdomains in the DG spatial discretization is achieved by the definition of the numerical flux at shared boundaries. DG lends itself to a simple computation–communication overlap; though CG can benefit from computation–communication overlap as well, it requires more effort to do so (Deville et al., 2002). Overlapping is especially important in a multi-GPU implementation to hide the latency associated with the data transfer between the CPU and GPU. Inter-processor flux calculation requires values from the left and right elements sharing a face; however, intra-processor flux calculation and computation of volume integrals can proceed while the necessary communication for computing inter-processor flux is going on. Algorithm 9 shows an outline of a multi-GPU DG implementation with communication–computation overlap. The latest technology in GPUs allows for copying data asynchronously using streams. We overlap computation and communication using two streams designated for each. The copying of data to and from the GPU is carried out on the copy stream (COPY), all computations on the GPU are done on the computation stream (COMP), and MPI communications between CPUs are on the host stream (HOST). A wait statement invoked on any device stream blocks the host thread until all operations on that stream come to completion. Even though we do not show it for the sake of simplicity, the communication of $\nabla \mathbf{q}$ for the LDG stabilization method is also done similarly.

Algorithm 9 Asynchronous Multi-GPU DG.

Procedure Asynch_DG_Comm
[COMP] Pack boundary data to a contiguous block of global memory
[COMP] Wait
[COPY] Start copying boundary data asynchronously from GPU to CPU
[COMP] Start computing volume integrals and intra-processor flux
[COPY] Wait
[HOST] Send boundary data to neighboring processors asynchronously
[HOST] MPI_waitall
[COPY] Start copying boundary data asynchronously from CPU to GPU
[COPY] Wait
[COMP] Compute inter-processor flux

7 Performance tests

7.1 Speedup results

First, we present speedup results for the GPU implementation of NUMA against the base Fortran code.[3] In Table 3, the time to solution of three test cases,solved using explicit DG, is presented. This information is useful to get a rough estimate of the performance per dollar of NUMA on different GPU cards. The specification, peak single and DP GFLOPS/s and bandwidth in GB/s,for the different types of GPU cards used in this work are given in Table 4. The Error Correcting Code (ECC) feature in these NVIDIA GPUs is disabled for all performance tests.

We will present the details of the test cases later in section 8; here we give the workload of each problem:

1. 2D Rising-thermal bubble: 100 elements with polynomial order 7, for a total of 51,200 nodes;
2. 3D Rising-thermal bubble: 1000 elements with polynomial order 5, for a total of 216,000 nodes;
3. acoustic wave on the sphere: 1800 elements with polynomial order 4, for a total of 225,000 nodes.

where nodes, here, denote the number of gridpoints in the mesh. We obtained two orders of magnitude speedups on the newer GPU cards, GTX Titan Black and K20X, over a single-core Intel and AMD CPUs, respectively. The speedup on the relatively older Tesla C2070 GPU card is a modest $50\times$ for SP arithmetic, which may make it competitive in terms of the performance per dollar comparisons.

Next, we present performance tests on the Titan supercomputer located at the Oak Ridge National Laboratory, where each node is equipped with a K20X GPU card and a 16-core 2.2, GHz AMD Opteron 6274 CPU. The speedup results are reported relative to the base Fortran code using all 16 cores of the CPU. We will examine the different kernel design and parameter choices we made in section 5 using the 2D rising thermal bubble benchmark problem.

The problem size is increased progressively from $10 \times 10 = 100$ elements until we fill up all the memory available on the device at $160 \times 160 = 25,600$ elements. The first test result, presented in Table 5, evaluates the

Table 3 Speedup comparison between CPU and GPU for both SP and DP calculations. The test is conducted on three types of GPU cards: an old Tesla C2070 and two newer cards GTX Titan Black and K20X GPUs. The time to solution is given in seconds for the CPU/GPU along with relative speedup. Two orders of magnitude performance improvement is obtained relative to a single-core CPU with the newer cards.

Test case	Double precision			Single precision		
	CPU	GPU	Speedup	CPU	GPU	Speedup
Tesla C2070 GPU vs One core of Intel Xeon E5645						
2D rtb	930.1	27.8	33.4	612.3	13.4	45.6
3D rtb	4408.9	141.9	31.1	3097.0	54.5	56.8
Acoustic wave	3438.8	96.7	35.6	2379.9	44.4	53.6
GTX Titan Black GPU vs One core of Intel Xeon E5645						
2D rtb	930.1	8.87	104.9	612.3	4.67	131.0
3D rtb	4408.9	41.47	106.3	3097.0	18.68	165.8
Acoustic wave	3438.8	26.72	128.7	2379.9	15.56	
K20X GPU vs 16-cores of 2.2,GHz AMD Opteron 6274						
2D rtb	103.17	13.97	7.38	77.75	6.89	11.28
3D rtb	434.36	61.14	7.10	339.61	28.12	12.08
Acoustic wave	166.06	21.10	7.87	132.46	11.24	11.78

Table 4. Specs of the NVIDIA GPU cards used in this work.

GPU	SP TFLOPS/s	DP TFLOPS/s	GB/s
Tesla C2070	1.03	0.52	144
Tesla K20c	3.52	1.17	208
Tesla K20X	3.95	1.31	225
GTX Titan Black	5.12	1.71	336

performance of the cube volume kernel at low-order polynomials using both OpenCL and CUDA translations of the native OCCA code. Although NVIDIA hardware includes interfaces for both OpenCL and CUDA, we obtained better performance with CUDA kernels on this particular hardware. Also, we observe markedly better speedups at polynomial orders 4 and 7 compared to other polynomial orders. The reason for

Table 5. OpenCL vs CUDA: Speedup comparison between CPU and GPU for DP calculations at different numbers of elements and polynomial orders using OpenCL and CUDA translation of the native OCCA kernel code. The GPU card is K20X and the CPU is a 16-core 2.2,GHz AMD Opteron 6274. The timing (in s) and speedup are given first for OpenCL and then for CUDA. The results show CUDA compiled kernels are optimized better. Also polynomial orders 4 and 7 give better speedup numbers in all cases.

N	$10\times10 = 100$ elements			$30\times30 = 900$ elements			$40\times40 = 1600$ elements		
	CPU	GPU	Speedup	CPU	GPU	Speedup	CPU	GPU	Speedup
2	1.46	0.59/0.52	2.47/2.81	10.62	2.57/2.17	4.13/4.90	18.83	4.34/3.70	4.34/5.09
3	2.68	0.69/0.59	3.88/4.54	22.01	3.56/3.06	6.18/7.19	41.53	5.84/5.04	7.11/8.24
4	5.30	0.97/0.86	5.46/6.16	46.45	5.50/5.12	8.45/9.07	81.91	9.27/8.69	8.84/9.43
5	8.12	1.47/1.37	5.52/5.93	77.03	10.53/9.88	7.32/7.80	137.49	18.33/17.11	7.50/8.04
6	13.89	2.27/2.11	6.11/6.58	122.27	17.24/16.11	7.09/7.59	210.35	30.15/28.15	6.98/7.47
7	20.49	2.68/2.41	7.64/8.50	195.61	20.82/18.87	9.40/10.37	343.74	36.36/33.05	9.45/10.40
N	$80\times80 = 6400$ elements			$120\times120 = 14,400$ elements			$160\times160 = 25,600$ elements		
	CPU	GPU	Speedup	CPU	GPU	Speedup	CPU	GPU	Speedup
2	80.72	15.71/13.33	5.14/6.05	184.19	33.47/27.82	5.50/6.62	336.19	61.56/52.01	5.46/6.46
3	179.07	21.46/18.46	8.34/9.70	405.15	47.63/41.08	8.51/9.86	729.17	84.40/72.61	8.64/10.04
4	350.54	35.01/32.71	10.01/10.71	798.50	77.85/72.77	10.26/10.97	1392.60	138.64/129.64	10.04/10.74
5	587.17	71.90/67.03	8.17/8.76	1329.79	161.42/150.56	8.24/8.83	2352.46	286.74/267.48	8.20/8.79
6	925.25	118.81/110.92	7.79/8.34	2086.84	267.12/249.50	7.82/8.36	-	-	-
7	1406.61	142.67/130.16	9.86/10.81	3158.43	320.77/293.05	9.84/10.78	-	-	-

the good performance at polynomial order 7 is due to the thread block size of $(7 + 1)^3 = 512$ that perfectly fits the hardware block size. Polynomial order 4 gives a thread block size of 125, which is only slightly less than 128. Therefore, this observation emphasizes the importance of selecting parameters to get optimum block dimensions that are multiples of the warp size.

GPUs are known to deliver higher performance using SP arithmetic than DP. For instance, the SP peak performance of a K20X GPU is 3× more than its DP peak performance.In Table 6, we present the speedup results comparing SP and DP performance. We obtain a maximum speedup of about 15× and 11× using SP and DP calculations, respectively. The reason for the different speedup numbers for SP and DP is that NUMA running on the CPU is able to achieve a speedup of only 1.5× using SP, while the GPU performance more than doubles using SP.

For low-order polynomials, we can process two or more elements per block to get an optimal block size. Table 7 shows the performance comparison of this scheme using one and two elements per block. We can see that the performance is significantly improved by processing two elements per block for up to polynomial order five; the block size, when processing two elements per block, exceeds the hardware limit at polynomial orders above five. The 100 elements simulation is not able to see any benefit from this approach because the device will not be fully occupied when processing two elements per block. All the other runs show significant benefits from processing two elements per block, except at polynomial order four — for which performance remains more or less the same. We mentioned earlier that polynomial order four gives a block size that is close to optimal, hence, there is really no need to process more than one element per block for this particular configuration.

We mentioned in section 5 that using vector datatype *float4* to store field variables may help to improve performance because one load operation is issued when fetching a *float4* data instead of four. Table 8 compares the speedup obtained using *float1* and *float4* versions of the volume kernel. The *float4* version performs better in most of the cases; here, again, the performance at polynomial order four is more or less the same.

We discussed in section 5 different ways to handle the problem with hardware limitations for high-order polynomial approximations. Thread block size and shared memory hardware limits allow us to use the volume kernel we tested so far up to polynomial order seven. First, we compare the performance of the four ways to use shared and L1 cache memory; namely, the naïve, Shared-1, Shared-2 and two-pass (horizontal + vertical) methods. Figure 3 shows that the two-pass method performs the best — about two times better than the naïve approach, which does not use shared

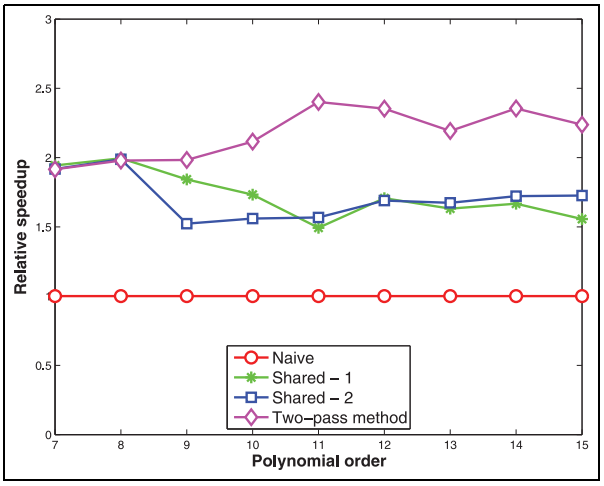

Figure 3. Comparison of different ways of exploiting the fast L1 cache and Shared memory in the volume kernel. The speedups obtained in DP arithmetic are reported relative to the naïve approach. The two-pass method performs the best due to better use of shared memory. The GPU card is K20X and the CPU is a 16-core 2.2, GHz AMD Opteron 6274.

memory but totally relies on L1 cache. The Shared-1 and Shared-2 methods perform similarly; this implies that the Shared-2 approach suggested in Micikevicius (2009) is not working as expected. Even though we try to store the data in the vertical direction in registers, most of it spills to global thread private memory. Because the polynomial order is high and we are loading all field data (eight floats) to registers, the register pressure is too high for the method to show any benefit.

In Table 9, we present the performance of the high-order volume kernel that uses the two-pass method for polynomial orders 8 to 15. It is not possible to solve bigger size problems than 40 × 40 elements with polynomial order 15 on this GPU because of the limited memory of 6 GB per card. We get a maximum speedup of about 9× at higher order polynomials, which is slightly less than the 11× performance we obtained at low-order polynomials; this is understandable because the two-pass method loads data twice and performs calculations twice as well.

7.2 Individual kernel performance tests

To evaluate the performance of individual kernels, we measure the rate of floating point operations in GFLOPS/s and data transfer rate (bandwidth) in GB/s. Many GPU applications tend to be memory-bound, hence bandwidth is as important a metric as the rate of floating point operations.The results obtained will guide us how to go about optimizing kernel performance by classifying them as either compute-bound or memory-bound. A convenient visualization is the roofline model (Williams et al.,2009), which sets an upper bound on kernel performance based on peak GFLOPS/s and GB/s of the device.We use two approaches to

Table 6. Double vs single precision: Speedup comparison between CPU and GPU for SP and DP calculations at different numbers of elements and polynomial orders using a CUDA translation of OCCA kernel code. A maximum speedup of about 15× is observed. The CPU/GPU times and speedups are given first for DP and then for SP. The GPU card is K20X and the CPU is a 16-core 2.2,GHz AMD Opteron 6274.

N	$10\times10 = 100$ elements			$30\times30 = 900$ elements			$40\times40 = 1600$ elements		
	CPU	GPU	Speedup	CPU	GPU	Speedup	CPU	GPU	Speedup
2	1.46/1.39	0.52/0.47	2.81/2.96	10.62/9.98	2.17/1.57	4.90/6.36	18.83/17.41	3.70/2.53	5.09/6.88
3	2.68/2.60	0.59/0.49	4.54/5.31	22.01/19.66	3.06/1.87	7.19/10.51	41.53/34.72	5.04/3.06	8.24/11.35
4	5.30/4.51	0.86/0.54	6.16/8.35	46.45/35.19	5.12/3.03	9.07/11.61	81.91/63.55	8.69/5.07	9.43/12.53
5	8.12/7.23	1.37/0.77	5.93/9.39	77.03/61.35	9.88/4.86	7.80/12.62	137.49/107.30	17.11/8.35	8.04/12.85
6	13.89/11.18	2.11/1.07	6.58/10.45	122.27/95.67	16.11/7.71	7.59/12.41	210.35/166.40	28.15/13.49	7.47/12.33
7	20.49/15.97	2.41/1.31	8.50/12.19	195.61/135.21	18.87/9.65	10.37/14.01	343.74/236.09	33.05/16.86	10.40/14.00
N	$80\times80 = 6400$ elements			$120\times120 = 14,400$ elements			$160\times160 = 25,600$ elements		
	CPU	GPU	Speedup	CPU	GPU	Speedup	CPU	GPU	Speedup
2	80.72/70.41	13.33/8.94	6.05/7.88	184.19/172.85	27.82/19.78	6.62/8.74	336.19/285.83	52.01/34.92	6.46/8.18
3	179.07/142.19	18.46/11.18	9.70/12.72	405.15/324.78	41.08/24.87	9.86/13.06	729.17/589.22	72.61/44.10	10.04/13.36
4	350.54/268.69	32.71/19.02	10.71/14.13	798.50/599.25	72.77/42.34	10.97/14.15	1392.60/1069.24	129.64/76.01	10.74/14.07
5	587.17/429.66	67.03/32.38	8.76/13.27	1329.79/1007.31	150.56/72.08	8.83/13.97	2352.46/1729.34	267.48/129.28	8.79/13.37
6	925.25/696.25	110.92/52.91	8.34/13.16	2086.84/1586.54	249.50/118.39	8.36/13.40	-	-	-
7	1406.61/968.10	130.16/66.41	10.81/14.58	3158.43/2227.29	293.05/148.76	10.78/14.97	-	-	-

Table 7. Multiple elements per block: The performance of the cube volume kernel can be improved by processing more than one element in a thread block simultaneously. The GPU times and speedups are given first for the one- element-per-block and then for the two-elements-per-block approaches. Improvement in performance is observed using two-elements-per-block in all the cases except for the 10×10 elements case, which does not fully occupy the GPU device when processing two-elements-per-block. The GPU card is K20X and the CPU is a 16-core 2.2,GHz AMD Opteron 6274.

N	$10\times10 = 100$ elements			$30\times30 = 900$ elements			$40\times40 = 1600$ elements		
	CPU	GPU	Speedup	CPU	GPU	Speedup	CPU	GPU	Speedup
2	1.46	0.52/0.57	2.81/2.56	10.62	2.17/1.81	4.90/5.87	18.83	3.70/2.85	5.09/6.61
3	2.68	0.59/0.61	4.54/4.39	22.01	3.06/2.93	7.19/7.51	41.53	5.04/4.74	8.24/8.76
4	5.30	0.86/0.92	6.16/5.76	46.45	5.12/5.74	9.07/8.09	81.91	8.69/9.81	9.43/8.35
5	8.12	1.37/1.37	5.93/5.92	77.03	9.88/9.68	7.80/7.96	137.49	17.11/16.72	8.04/8.22
N	$80\times80 = 6400$ elements			$120\times120 = 14,400$ elements			$160\times160 = 25,600$ elements		
	CPU	GPU	Speedup	CPU	GPU	Speedup	CPU	GPU	Speedup
2	80.72	13.33/9.96	6.05/8.10	184.19	27.82/21.10	6.62/8.73	336.19	52.01/38.09	6.46/8.83
3	179.07	18.46/17.5	9.70/10.23	405.15	41.08/38.51	9.86/10.52	729.17	72.61/67.62	10.04/10.78
4	350.54	32.71/37.15	10.71/9.43	798.50	72.77/82.93	10.97/9.63	1392.60	129.64/147.61	10.74/9.43
5	587.17	67.03/65.2	8.76/9.00	1329.79	150.56/146.67	8.83/9.07	2352.46	267.48/260.89	8.79/9.02

Table 8. *float1 vs float4*: Theeffect of using *float4* for computing the gradientin the volume kernel is compared against the version of the volume kernel where one field variable is loaded. The CPU/GPU time and speedups are given first for *float1* and then for *float4*. Some improvement is observed in most cases except when using polynomial order four, which results in a good thread block size. The GPU card is K20X and the CPU is a 16-core 2.2,GHz AMD Opteron 6274.

N	$10\times10 = 100$ elements			$30\times30 = 900$ elements			$40\times40 = 1600$ elements		
	CPU	GPU	Speedup	CPU	GPU	Speedup	CPU	GPU	Speedup
2	1.46	0.52/0.47	2.81/3.11	10.62	2.17/2.06	4.90/5.15	18.83	3.70/3.33	5.09/5.65
3	2.68	0.59/0.57	4.54/4.70	22.01	3.06/3.10	7.19/7.10	41.53	5.04/5.14	8.24/8.08
4	5.30	0.86/0.82	6.16/6.46	46.45	5.12/5.10	9.07/9.11	81.91	8.69/8.69	9.43/9.43
5	8.12	1.37/1.27	5.93/6.39	77.03	9.88/9.38	7.80/8.21	137.49	17.11/16.29	8.04/8.44
6	13.89	2.11/1.93	6.58/7.19	122.27	16.11/14.86	7.59/8.23	210.35	28.15/26.06	7.47/8.07
N	$80\times80 = 6400$ elements			$120\times120 = 14,400$ elements			$160\times160 = 25,600$ elements		
	CPU	GPU	Speedup	CPU	GPU	Speedup	CPU	GPU	Speedup
2	80.72	13.33/12.00	6.05/6.73	184.19	27.82/26.50	6.62/6.95	336.19	52.01/46.85	6.46/7.18
3	179.07	18.46/18.99	9.70/9.43	405.15	41.08/42.66	9.86/9.50	729.17	72.61/74.93	10.04/9.73
4	350.54	32.71/32.88	10.71/10.66	798.50	72.77/73.00	10.97/10.94	1392.60	129.64/129.72	10.74/10.74
5	587.17	67.03/64.12	8.76/9.16	1329.79	150.56/144.01	8.83/9.23	2352.46	267.48/256.62	8.79/9.17
6	925.25	110.92/102.85	8.34/9.00	2086.84	249.50/222.08	8.36/9.40	-	-	-

determine the GFLOPS/s and GB/s: hand-counting the number of floating point operations and bytes loaded to get an *estimate* of the arithmetic throughput and bandwidth, and using a profiler to get the *effective* values.

The first results, shown in Figures 4a–4d, are produced by hand-counting the number of FLOPS and bytes loaded from global memory per kernel execution. This would be enough to calculate the arithmetic intensity (GFLOPS/GB) and determine whether a kernel would be memory- or compute- bound; however, we need to conduct actual simulations to determine kernel execution time and, thus, the efficiency of our kernels in terms of GFLOPS/s and GB/s. The roofline plots show that our efficiency increases with problem size and reaches about 80% for the volume and surface kernels, while 100% efficiency is observed for the update and project kernels. These tests are conducted on the isentropic vortex problem (see section 8.1), which concerns advection of a vortex by a constant velocity. The GPU is a Tesla K20c GPU card.

The highest rate of floating point operations observed in any of the CG or DG kernels is about 320, GFLOPS/s for the horizontal volume kernel at $N = 10$ using SP arithmetic. The vertical volume kernel is a close second, but the surface and update kernels lag far

Table 9. Higher order polynomials: The performance of the two-pass method, with horizontal + vertical split, is evaluated at higher order polynomials in DP calculations. This kernel performs slower than the cube volume kernel when used for low-order polynomials, but it is the best performing version among the volume kernels we considered for high order. The GPU card is K20X and the CPU is a 16-core 2.2,GHz AMD Opteron 6274.

N	$10\times10 = 100$ elements			$30\times30 = 900$ elements			$40\times40 = 1600$ elements		
	CPU	GPU	Speedup	CPU	GPU	Speedup	CPU	GPU	Speedup
8	31.17	4.77	6.53	271.50	33.19	8.18	492.23	58.17	8.46
9	43.21	5.90	7.32	373.63	44.52	8.39	666.37	77.84	8.56
10	59.89	7.14	8.38	493.54	55.02	8.97	909.75	96.49	9.42
11	79.86	9.61	8.31	691.65	75.52	9.15	1199.67	132.28	9.07
12	103.64	13.40	7.73	923.22	107.44	8.59	1713.01	190.06	9.01
13	131.74	16.89	7.80	1140.64	138.13	8.25	2009.99	243.28	8.26
14	169.49	23.52	7.20	1468.72	195.32	7.52	2568.77	340.99	7.53
15	220.91	28.36	7.79	1862.14	233.06	7.99	3352.22	410.42	8.17

behind in terms of GFLOPS/s performed. The update kernel, which does the explicit RK time integration, shows the highest bandwidth performance at about 208 GB/s, which is in fact the peak memory bandwidth of the device. The project kernel, which does the scatter–gather operation of CG, comes in a close second. The volume and surface kernels, though they have the highest arithmetic intensity, lag behind in terms of bandwidth performance. Therefore, no single kernel exhibits best performance in terms of both GFLOPS/s and bandwidth.

The roofline plots expose that the arithmetic intensity (GFLOPS/GB) of the update kernel, project kernel and surface kernel do not change with polynomial order. When extrapolated, all three vertical lines hit the diagonal of the roofline, confirming the fact that these kernels are memory-bound. The arithmetic intensity of the volume kernels increases with polynomial order, complicating the classification to either compute- or memory-bound; however, with polynomial degree up to 11, the kernels are still well within the memory-bound region.

The second group of kernel performance tests, shown in Figure 5a–5d, were conducted using a GTX Titan Black GPU. For these tests we used *nvprof*, to determine the effective arthimetic throughput and memory bandwidth. As a result, the plots obtained from this test are less smooth than the previous plots, which were produced by hand-counting the FLOPS and GB of kernels. Moreover, here we use the cube volume kernels instead of the split horizontal + vertical kernels. We also changed the test case to a 2D rising thermal bubble problem, which requires numerical stabilization, to invoke the diffusion kernel. The highest GFLOPS/s observed in this test was 700 GFLOPS/s for the volume kernel using SP. To compare performance with the previous tests that were produced using a different GPU, we look at the roofline plots instead. We expect the roofline plot for the combined volume kernel to lean more towards the compute-bound region because more floating point operations are done per byte of data loaded. Indeed this turns out to be the case even though the cube volume kernels were run up to a maximum polynomial order of eight. The diffusion kernels, used for computing the Laplacian, also show similar performance characteristics to the volume kernels.

7.3 Scalability test

The scalability of the multi-GPU implementation was tested on a GPU cluster, namely, the Titan supercomputer, which has 18,688 NVIDIA Tesla K20X GPU accelerators. We conduct a weak scalability test, where each GPU gets the same workload, using the 2D rising thermal bubble problem discussed in section 8.2, using 900 elements per GPU with polynomial order 7 in all directions. In a weak scaling test, the time to solution should, ideally, stay constant as the workload is increased; however, delays are introduced due to the need for communication between GPUs.The scalability result in Figure 6 shows that the GPU version of NUMA is able to achieve 90% scaling efficiency on tens of thousands of GPUs. Different implementations of the unified CG/DG algorithms are tested, among which, DG with overlapping of computation and communication to hide latency performed the best. Our current CG implementation does not overlap communication with computation and, as a result, its scalability suffers.

The 900 element grid per GPU used for producing the scalability plot is far from filling up the GPU memory, hence, the scalability could be improved by increasing the problem size further. We compare scalability up to 64 GPUs, which is the point where the efficiency of the parallel implementation flattens out, for different number of elements in Figure 8. The scalability increases by more than 20% going from a 100 to 900 elements grid per GPU.

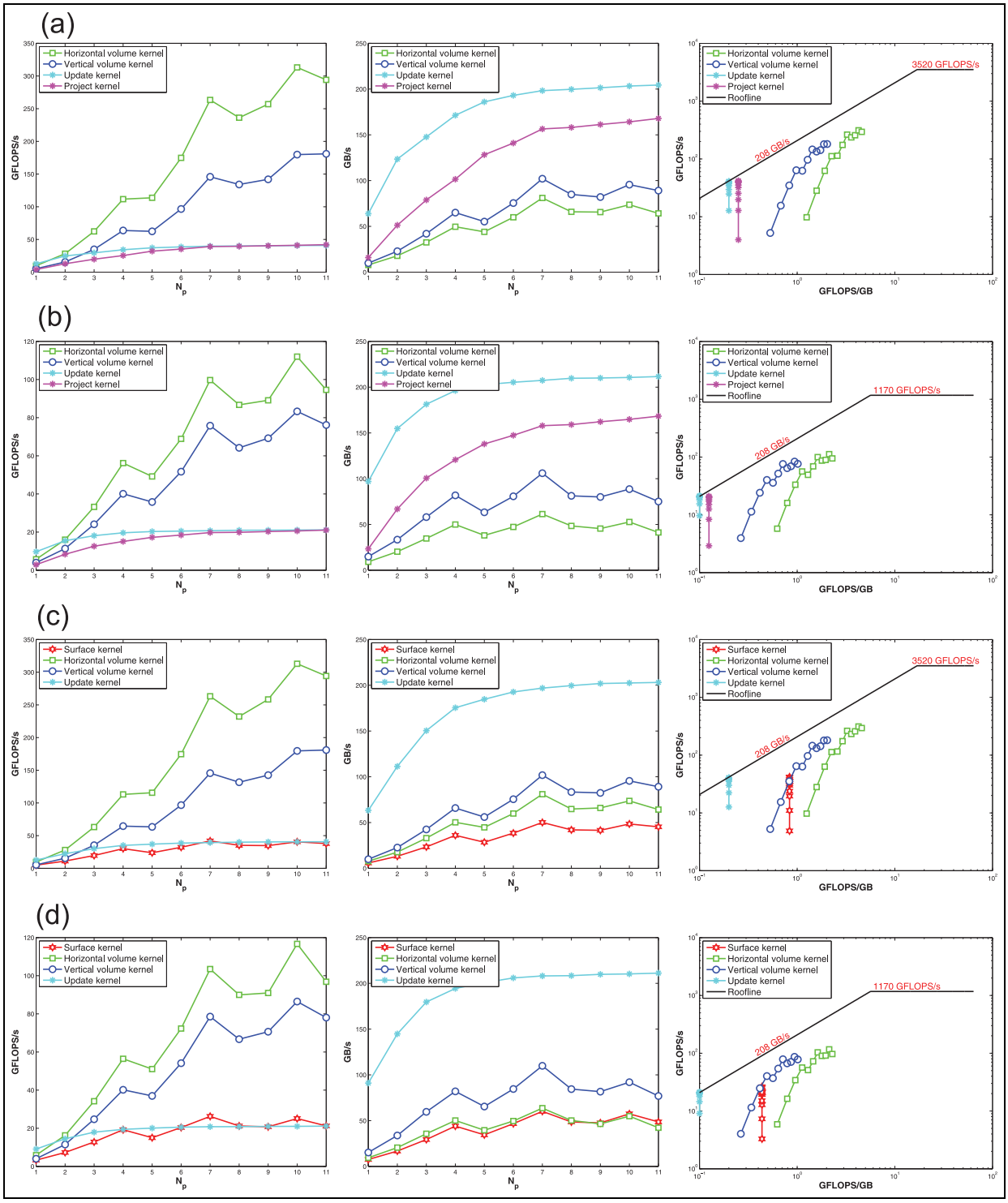

Figure 4. Performance of individual kernels: The efficiency of our kernels was tested on a mini-app developed for this purpose.The FLOPS and bytes for this test are counted manually. The volume kernel, which is split into two (horizontal + vertical), has the highest rate of FLOPS/s. The timestep update kernel has the highest bandwidth usage at 208, GB/s.The SP and DP performance of the main kernels in CG and DG are shown in terms of GFLOPS/s, GB/s and roofline plots to illustrate their efficiency. The GPU is a Tesla K20c. (a) SP-CG kernels performance. (b) DP-CG kernels performance. (c) SP-DG kernels performance. (d) DP-DG kernels performance.

In operational NWP, strong scaling on multi-GPU systems may be as important as weak scaling because of limits placed on the simulation time to make a day's weather forecast. For this reason, we also conducted strong scaling tests, shown in Figure 7, on a global scale simulation problem described in section 8.5. Our

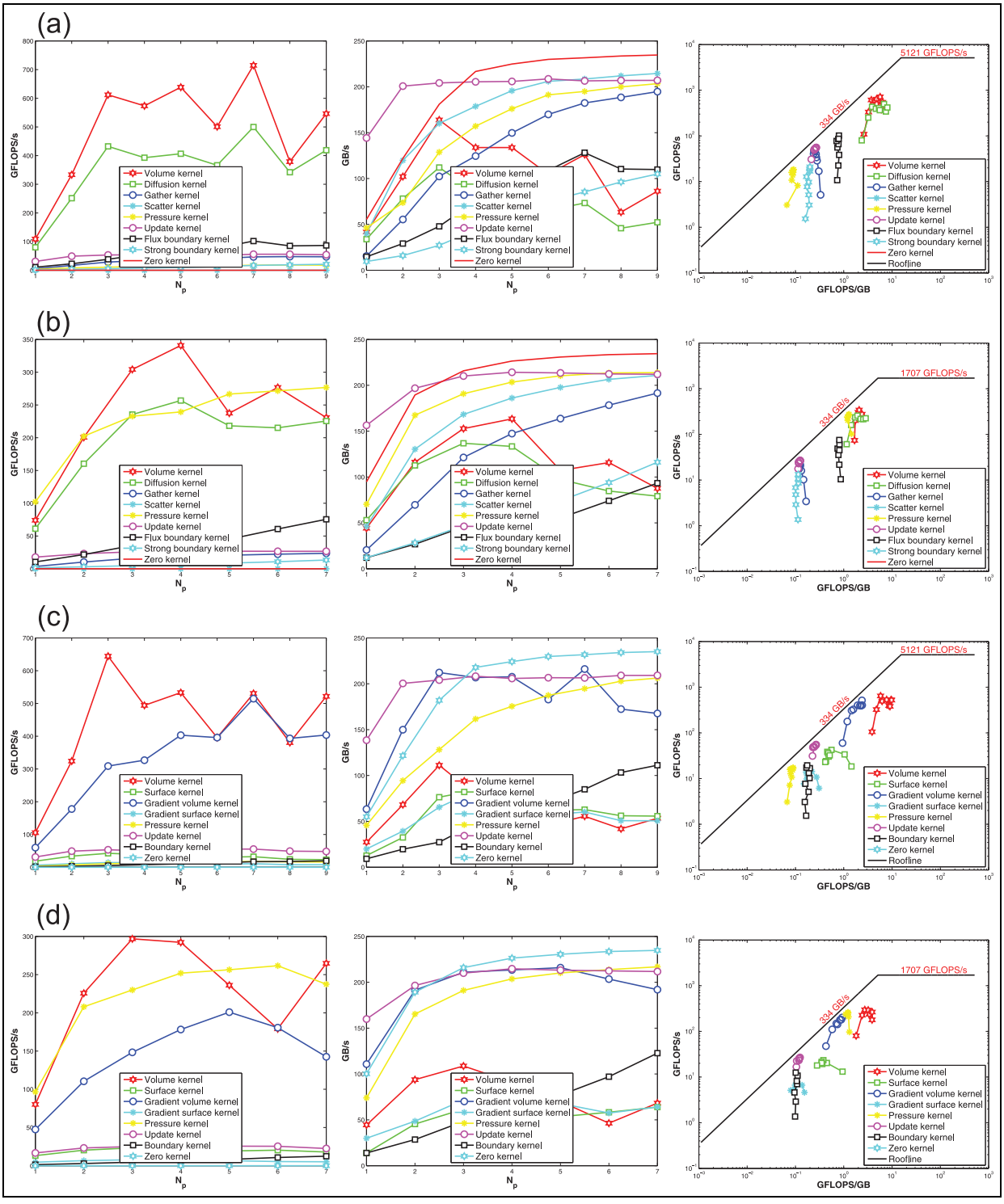

Figure 5. Performance of individual kernels: The efficiency of our kernels was tested after being incorporated to the base NUMA code. The measurements for this test were done using *nvprof*: effective memory bandwidth = *dram_read_throughput* + *draw_write_throughput*, and effective arithmetic throughput = *flop_dp/sp_efficiency*. The SP and DP performance of the main kernels in CG and DG are shown in terms of GFLOPS/s, GB/s and roofline plots to illustrate their efficiency. The GPU is a GTX Titan Black. (a) SP-CG kernels performance. (b) DP-CG kernels performance. (c) SP-DG kernels performance. (d) DP-DG kernels performance.

goal here was to determine the number of GPUs required for a given simulation time limit for different resolutions, namely, coarse grids of 13 km and 10 km resolutions and a finer grid with 3 km resolution. The grids are cubed spheres with $6 \times 112 \times 112 \times 4$, $6 \times 144 \times 144 \times 4$, and $6 \times 448 \times 448 \times 4$

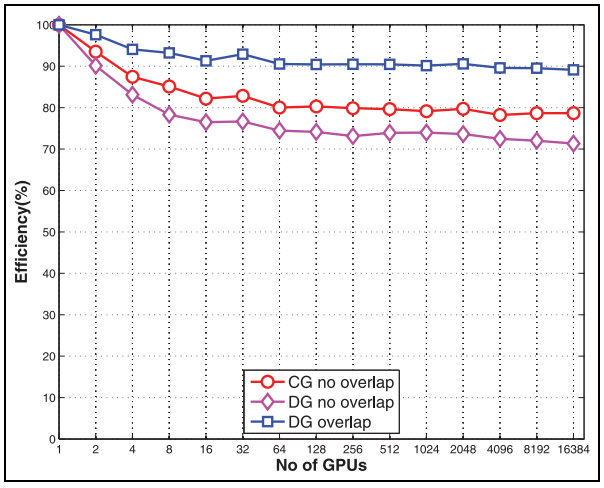

Figure 6. Weak scalability test of multi-GPU implementation of NUMA: The scalability of NUMA for up to 16,384 GPUs on the Titan supercomputer is shown. Each node of Titan contains a Tesla K20X GPU. An efficiency of about 90% is observed relative to a single GPU. The test was conducted using a unified implementation of CG and DG. The efficiency of DG was significantly improved (by about 20%) when overlapping communication with computation, which helps to hide both the data copying latency between CPU and GPU and CPU–CPU communication latency.

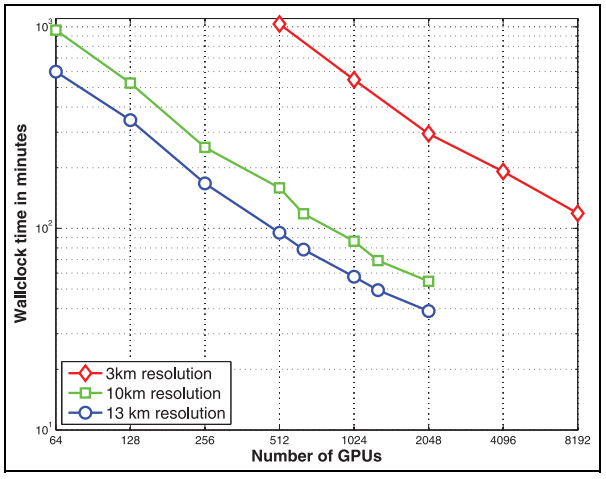

Figure 7. Strong scalability tests on a global scale simulation using grid resolutions of 13 km ($6 \times 112 \times 112 \times 4$ elements at $N = 7$), 10 km ($6 \times 144 \times 144 \times 4$ elements at $N = 7$) and 3 km ($6 \times 448 \times 448 \times 4$ elements at $N = 7$).The number of GPUs required to fulfill a target 100 min wallclock time for a one-day forecast is about 512, 1024 and 8192,GPUs for the 13 km, 10 km, and 3 km resolutions, respectively.

elements[4] for the 13 km, 10 km and 3 km resolutions, respectively, using a polynomial order of $N = 7$. The plot shows that about 512, 1024 and 8192,GPUs are required to bring down the simulation time below 100 min for grid resolutions of 13 km, 10 km and 3 km, respectively. The workload on each GPU card to

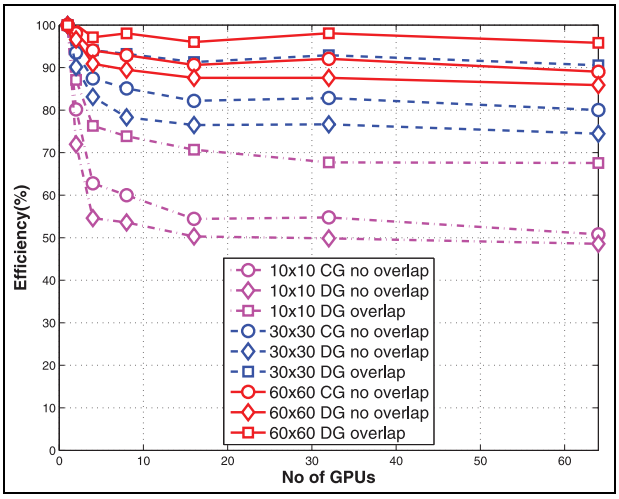

Figure 8. Weak scalability test of the multi-GPU implementation for different numbers of elements using up to 64 nodes of Titan. The 60×60 element grid gives a much better scalability than the 10×10 grid, hence, we expect better weak scalability results than shown in Figure 6 with bigger problem sizes.

meet the 100 min wallclock target was then calculated to be about 301 kNodes, 248 kNodes, and 301 kNodes for the 13 km, 10 km and 3 km resolutions, respectively. The 10 km resolution had a smaller kNodes per GPU because we actually met a lower target of about 85,min at 1024 GPUs. We believe that once we port the IMEX time integrators to the GPU, we can meet simulation time limits with much fewer GPUs than the 3 million CPU threads required to meet a 4.5 min wallclock time limit required using the CPU version of NUMA (see Müller et al. (2016) for details).

8 Validation with benchmark problems

The GPU implementation of our Euler solver was validated using a suite of bench mark problems showcasing various characteristics of atmospheric dynamics. We consider problems of different scale: cloud-resolving (microscale), limited area (mesoscale) and global scale atmospheric problems. Most of these test cases do not have analytical solutions against which comparisons can be made. For this reason, we first consider a rather simple test case of advection of a vortex by a uniform velocity, which has an analytical solution that will allow us to compute the exact L^2 error and establish the accuracy of our numerical model. The rest of the test cases serve as a demonstration of its application to practical atmospheric simulation problems.

8.1 2D Isentropic vortex problem

We begin verification with a simple test case that has an exact solution to the Euler equations. The test case

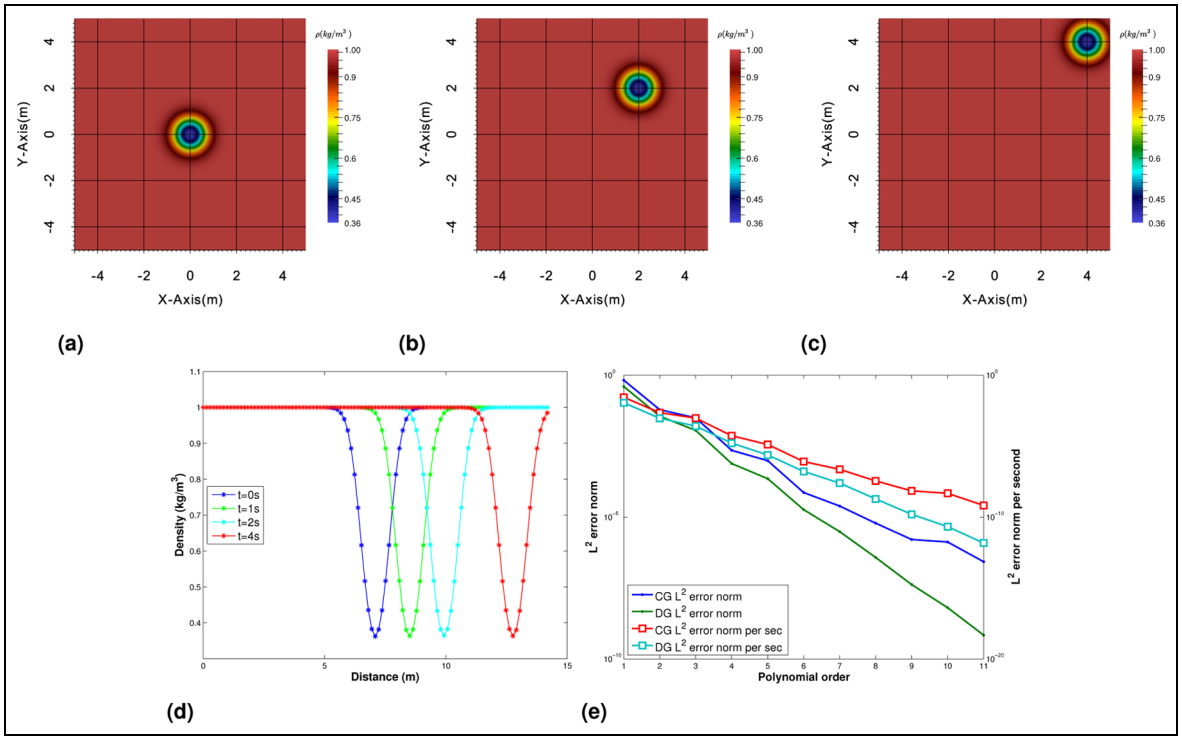

Figure 9. Isentropic vortex: A plot of density (ρ) of the vortex at different times shows that the vortex, traveling at a speed of 1 m/s, reaches the expected grid locations at all times. The density distribution within the vortex is maintained as shown in Figure 9d. A grid of $22 \times 22 \times 2$ elements with a 7th degree polynomial is used. (a) $t = 0$, s. (b) $t = 2$, s. (c) $t = 4$, s. (d) Density along diagonal. (e) L^2 error norm of density.

involves advective transport of an inviscid isentropic vortex in free stream flow. The problem is often used to test the ability of numerical methods to preserve flow features, such as vortices, in free stream flow for long durations.However, the problem is linear, and hence not suitable for testing the coupling of wave motion and advective transport that are the causes of nonlinearity in the Euler equations.

The free stream conditions are

$$\rho = 1, u = U_\infty, v = V_\infty, \theta = \theta_\infty$$

Perturbations are added in such a way that the flow is isentropic. The initial conditions are

$$(u', v') = \frac{\beta}{2\pi}\exp\left(\frac{1-r^2}{2}\right)(-y + y_c, x - x_c)$$

$$\theta = \theta_\infty - \frac{(\gamma-1)\beta^2}{8\gamma\pi^2}\exp(1-r^2)$$

where

$$r = \sqrt{(x-x_c)^2 + (y-y_c)^2}$$

We simulate the isentropic vortex problem on a $[-5\text{ m}, 5\text{ m}] \times [-5\text{ m}, 5\text{ m}] \times [-0.5\text{ m}, 0.5\text{ m}]$ computational domain, with $(x_c, y_c, z_c) = (0, 0, 0)$, $\beta = 5$, $U_\infty = 1$ m/s, $V_\infty = 1$ m/s and $\theta_\infty = 1$. The domain is subdivided into $22 \times 22 \times 2$ elements with a polynomial order of $N = 7$ in all directions for a total of about 0.5,million nodes. The simulation is run for 10 s with a constant timestep of $\Delta t = 0.001$ s using the modified RK time integration scheme discussed in section 4. We anticipate the vortex to move along the diagonal at a constant velocity while maintaining its shape. This is indeed what is obtained, as shown in Figure 9.

To evaluate the accuracy of the numerical model, we compute the L^2 norm of the error $\mathbf{q} - \mathbf{q}^\infty$ over the domain Ω_e, i.e. $||\mathbf{q} - \mathbf{q}^\infty||_{L^2(\Omega_e)}$, for both SP and DP arithmetic, where $\mathbf{q}^\infty$ is the exact solution.

The DP run takes about 267 s to complete while the SP run takes 161 s; however, the maximum error associated with the SP calculations is much larger, as shown in Figure 9e. Therefore, if this reduction in accuracy is acceptable for a certain application, then using SP arithmetic on the GPU is recommended.For this particular problem, DG gives a lower maximum error than CG in both the SP and DP calculations. The L^2-error of density decreases with increasing polynomial order as shown in Figure 9e; the per-second L^2-error also shows the same behavior, affirming the fact that higher order polynomials require less work per degree of freedom. $N = 11$ is the maximum polynomial order that we were able to run before we run out of global memory on the GPU.

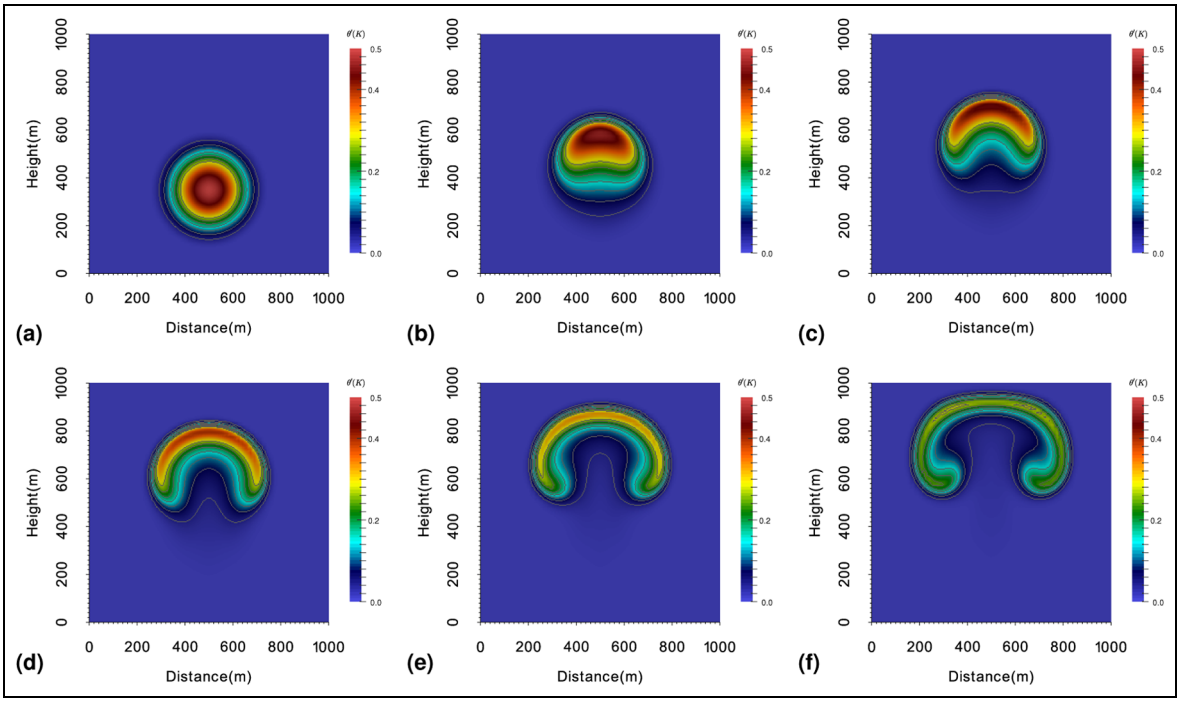

Figure 10. Potential temperature perturbation $\theta'(K)$ contour plot for the2D rising thermal bubble problemrun with CG and an artificial viscosity of $\mu = 1.5$ m^{2}s for stabilization. Results are shown at (a) $t = 0$, (b) 300, (c) 500, (d) 600, (e) 700 and (f) 900 s. A grid of 10 × 1 × 10 elements with 6th degree polynomials is used.

8.2 2D Rising thermal bubble

A popular benchmark problem in the study of non-hydrostatic atmospheric models is the 2D rising thermal bubble problem, first proposed in Robert (1993). The test case concerns the evolution of a warm bubble in a neutrally stratified atmosphere of constant potential temperature θ_0. The bubble is lighter than the surrounding air, hence, it rises while deforming due to the shear induced by the uneven distribution of temperature within the bubble. This deformation results in a mushroom-like cloud. The initial conditions for this test case are in hydrostatic balance in which pressure decreases with height as

$$p = p_0\left(1 - \frac{gz}{c_p\theta_0}\right)^{c_p/R}$$

The potential temperature perturbation isgiven by

$$\theta' = \begin{cases} 0 & \text{for } r > r_c \\ \frac{\theta_c}{2}\left(1 + \cos\left(\frac{\pi r}{r_c}\right)\right) & \text{for } r \le r_c \end{cases} \quad (18)$$

where

$$r = \sqrt{(x - x_c)^2 + (z - z_c)^2}$$

The parameters for the problem are similar to that found in Giraldo and Restelli (2008) and Ullrich and Jablonowski (2012): a domain of size [0 m, 1000 m] × [0,m, 100 m] × [0 m, 1000 m], with $(x_c, z_c) = (500\text{ m}, 350\text{ m})$, $r_c = 250$ m, and $\theta_c = 0.5$, K, $\theta_0 = 300$, K and an artificial viscosity of $\mu = 0.8$, m^{2}s for stabilization. The domain is subdivided into 10 × 1 × 10 elements with polynomial order $N = 6$ set in all directions for a total of about 180k nodes. The grid resolution is about 25,m therefore this problem can be considered as cloud resolving. An inviscid wall boundary condition is used on all sides.

The simulation is run for 1000 s using the explicit RK time integration method discussed in section 4 with aconstant timestep of $\Delta t = 0.02$ s. The status of the bubble at different times is shown in Figure 10. The results agree with those reported in Giraldo and Restelli (2008). Most importantly, the results are identical with those obtained using the CPU version of NUMA, even though those are not shown here. We should mention here that matching the CPU version of NUMA up to machine precision (e.g. 10^{-15}) has been an important goal in the development of the GPU code.

8.3 2D Colliding thermal bubbles

Next, we consider the case of colliding thermal bubbles proposed in Robert (1993). The shape of the rising warm bubble is now affected by the presence of a smaller sinking cold bubble on the RHS. This destroys the symmetry of the rising bubble. We should note here that the rising thermal bubble problem in section 8.2

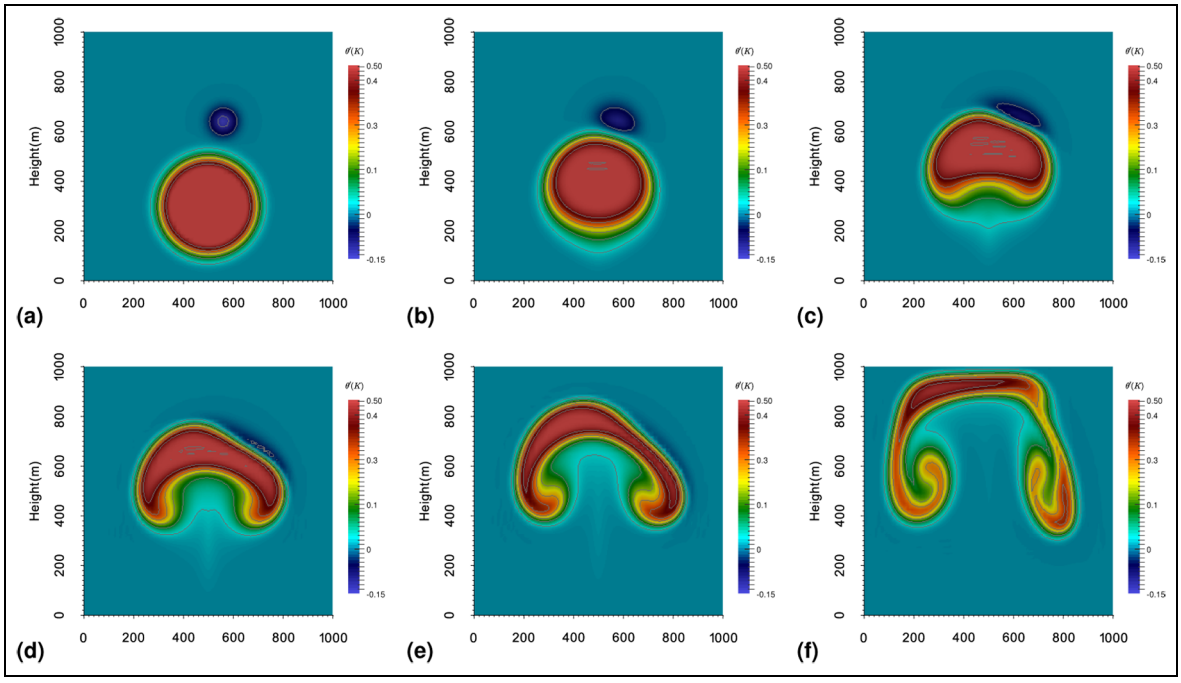

Figure 11. Colliding thermal bubbles. Evolution of potential temperature perturbation $\theta'(K)$ run with CG and an artificial viscosity of $\mu = 1.5$ m^{2}s for stabilization. Results are shown at (a) $t = 0$, (b) 300, (c) 500, (d) 600, (e) 700 and (f) 900,s. A grid of $10 \times 1 \times 10$ elements with 6th degree polynomials is used.

could have been solved considering only half of the domain because of symmetry, which is not the case here. Also, the potential temperature perturbation θ' is specified differently for this problem. Within a certain radius r_c, the perturbation is a constant θ_c; outside of this inner domain, it is defined by a Gaussian profile as

$$\theta' = \begin{cases} \theta_c & \text{for } r \leq r_c \\ \theta_c \exp[-((r-a)/s)^2] & \text{for } r > r_c \end{cases}$$

The warm bubble is centered at $(x_c, z_c) = (500\text{ m}, 300\text{ m})$, with a perturbation potential temperature amplitude of $\theta_c = 0.5$, radius $a = 150$ m and $s = 50$ m. The initial conditions for the cold bubble are: $(x_c, z_c) = (560\text{ m}, 640\text{ m})$, $\mu = 0.8$ m^{2}s, $\theta_c = 0.5$, $a = 0$, m and $s = 50$ m.

The result of the simulation is shown in Figure 11, which confirms the fact that the rising bubble indeed loses its symmetry. The edge of the rising bubble becomes sharper in some places from 600,s onwards. Qualitative comparison with the results shown in Robert (1993) and Yelash et al. (2014) show similar large-scale patterns, while small-scale patterns differ depending on the grid resolution used. Here, again the results of the CPU NUMA code are identical with the GPU version.

8.4 Density current

The density current benchmark problem, first proposed in Straka et al. (1993), concerns the evolution of a cold bubble in a neutrally stratified atmosphere of constant potential temperature θ_0. The dimensions of this test case are in the range of typical mesoscale models in which hydrostatic assumptions are valid. Because the bubble is colder than the surrounding air, it sinks and hits the ground, then moves along the surface while forming shearing currents, which then generate Kelvin–Helmholtz rotors. The numerical solution of this problem using high-order methods often requires the use of artificial viscosity or other methods for stabilization. We use a viscosity of $\mu = 75$ m^{2}s according to Straka et al. (1993).

The problem setup issimilar to that of the rising thermal bubble test case with the following differences:a cold bubble with $\theta_c = -15$ K in equation (18), a domain of $\Omega_e = [0, 25600\text{ m}] \times [0, \infty][0, 6400\text{ m}]$, ellipsoidal bubble with radii of $(r_x, r_z) = (4000\text{ m}, 2000\text{ m})$ and centered at $(x_c, z_c) = (0, 3000\text{ m})$. The problem is symmetric, therefore, we only need to simulate half of the domain. The computational domain is subdivided into $128 \times 1 \times 32$ elements with polynomial order of $N = 4$ set in all directions. With this set of choices, the effective resolution of our model is 50,m. Inviscid wall boundary conditions are used at all sides.

Figure 12 shows the evolution of potential temperature of the bubble up to 900 s. The vortical structures formed at $t = 900$ s, namely three Kelvin–Helmholtz instability rotors, are similar to that shown in Straka et al. (1993) and Ullrich and Jablonowski (2012). The first rotor is formed near the leading edge of the density current at 300 s, then the second rotor develops at the

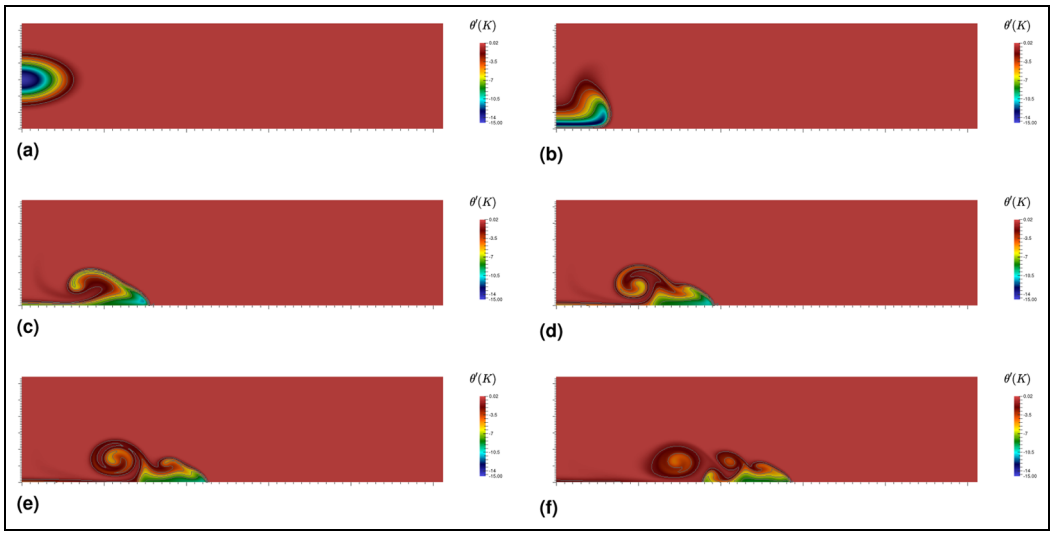

Figure 12. Density current. Evolution of potential temperature perturbation $\theta'(K)$ run with CG and an artificial viscosity of $\mu = 75$ m^{2}s for stabilization. Results are shown at (a) $t = 0$, (b) 300, (c) 600, (d) 700, (e) 800 and (f) 900 s. A grid of 128 × 1 × 32 elements with 4th degree polynomials is used for an effective resolution of 50 m in the x- and z-directions.

front of the density current around 600 s. Here again the GPU code matched results obtained using NUMA's CPU code, which has already been verified with many other atmospheric benchmark problems.

8.5 Acoustic wave

To validate the GPU implementation for global scale simulations on the sphere, we consider a test case of an acoustic wave traveling around the globe first described in Tomita and Satoh (2005). Several issues emerge that did not arise in the previous test cases. This test case validates 3D capabilities, curved geometry, metric terms, and a non-constant gravity vector. The initial state for this problem is hydrostatically balanced with an isothermal background potential temperature of $\theta_0 = 300$ K. A perturbation pressure P' is superimposed on the reference pressure

$$P' = f(\lambda, \phi)g(r)$$

where

$$f(\lambda,\phi) = \begin{cases} 0 & \text{for } r > r_c \\ \frac{\Delta P}{2}(1 + \cos(\frac{\pi r}{r_c})) & \text{for } r \le r_c \end{cases}$$

$$g(r) = \sin\left(\frac{n_v \pi r}{r_T}\right)$$

where $\Delta P = 100$ Pa, $n_v = 1$, $r_c = r_e/3$ is one third of the radius of the earth $r_e = 6371$ km and a model altitude of $r_T = 10$ km.The geodesic distance r is calculated as

$$r = r_e \cos^{-1}[\sin\phi_0 \sin\phi + \cos\phi_0 \cos\phi \cos(\lambda - \lambda_0)]$$

where (λ_0, ϕ_0) is the origin of the acoustic wave.

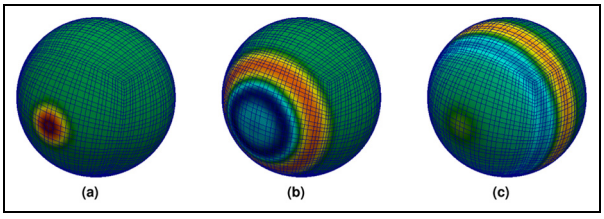

Figure 13. Propagation of an acoustic wave. The density perturbation after (a) 0 h, (b) 4 h and (c) 7 h. A cubed sphere grid with 6 × 10 × 10 × 3 elements with a 3rd degree polynomial is used.

The grid is a cubed sphere 6 × 10 × 10 × 3 for a total of 1800 elements with 3rd order polynomials.No-flux boundary conditions are applied at the bottom and top surfaces.Visual comparison of plots showing the location of the wave at different hours, shown in Figure 13, against results in Tomita and Satoh (2005) indicate that the results are quite similar to these results, as well as to those computed with the CPU version of NUMA.

The speed of sound is about $a = \sqrt{\gamma p/\rho} = 347.32$ m/s with the initial conditions of the problem. With this speed, the acoustic wave should reach the antipode in about 16 h. The result from the simulation indicates the acoustic wave has traveled 20.01 million meters within this time — which gives an average sound speed of 347.55 m/s, which is close to the calculated sound speed (a relative error of less than 1%).

9 Conclusions

In this work, we have ported the NUMA to the GPU and demonstrated speedups of two orders of magnitude

relative to a single-core CPU. Tests on one node of the Titan supercomputer, consisting of a K20X GPU and a 16-core AMD CPU, yielded speedups of up to $15\times$ and $11\times$ for the GPU relative to the CPU using SP and DP arithmetic, respectively. This performance is achieved by exploiting the specialized GPU hardware using suitable algorithms and optimizing kernels for performance.

NUMA solves the Euler equations using a unified continuous and discontinuous Galerkin approach for spatial discretization and various implicit and explicit time integration schemes. GPU kernels are written for different components of the dynamical core, namely, the volume integration kernel, surface integration kernel, (explicit) time update kernel, kernels for stabilization, etc. We use algorithms suitable for the Single Instruction Multiple Thread (SIMT) architecture of GPUs to maximize bandwidth usage and rate of floating point operations (FLOPS) of the kernels. Some of the kernels, for instance the volume integration, turned out to be high on the FLOPS side, while some others, such as the explicit time integration kernel, are high on bandwidth usage. Optimizations of kernels should be geared towards achieving the maximum attainable efficiency as bounded by the roofline model.

We have also implemented a multi-GPU version of NUMA using the existing MPI-infrastructure for multi-core CPUs (Kelly and Giraldo, 2012). Communication between GPUs is done via CPUs by first copying the inter-processor data from the GPU to the CPU. For the discontinuous Galerkin (DG) implementation, we overlap communication and computation to hide latency of data copying from the GPU and communication between CPUs.We then tested the scalability of our multi-GPU implementation using 16,384 GPUs of the Titan supercomputer — the third fastest supercomputer in the world as of June 2016. We obtained a weak scaling efficiency of about 90% that increases with bigger problem size. The CG and DG methods that do not overlap communication with computation performed about 20% less efficiently, thereby, highlighting the value of this approach.

For portability to a heterogeneous computing environment, we used a novel programming language called OCCA, which can be cross-compiled to either OpenCL, CUDA or OpenMP at runtime.Finally, the accuracy and performance of our GPU implementations were verified using several benchmark problems representative of different scales of atmospheric dynamics.

In the current work, we ported only the explicit time integration modules to the GPU. However, operational NWP often requires use of IMEX time integration to counter the limitation imposed by the Courant number. In the future, we plan to port the IMEX time integration modules, which require solving a system of equations at each timestep.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship and/or publication of this article: This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy (contract number DE-AC05-00OR22725). The authors gratefully acknowledge support from the Office of Naval Research (grant number PE-0602435N).

Notes

1. The gravity vector is constant in mesoscale models, whereas it varies with location in global scale models.
2. Hyper-diffusion can also be used, but in order to simplify the exposition, we shall only remark on second-order diffusion.
3. The base Fortran code is the original CPU code, i.e. the non-OCCA implementation that we use on the GPUs.
4. On cubed sphere grids, the total number of elements is denoted as $N_{\text{panels}} \times N_\xi \times N_\eta \times N_\zeta$ where $N_{\text{panels}} = 6$ for the six panels of the cubed sphere, $N_\xi = N_\eta$ and N_ζ are the number of elements in both horizontal directions and the vertical direction, respectively.

References

Abdi DS and Giraldo FX (2016) Efficient construction of unified continuous and discontinuous Galerkin formulations for the 3D Euler equations. *Journal of Computational Physics* 320: 46–68.

Allard J, Courtecuisse H and Faure F (2011) Implicit FEM solver on GPU for interactive deformation simulation. In: Mei W and Hwu W (eds) *GPU Computing Gems Jade Edition*. Boston: Elsevier, pp.281–294.

Bassi F and Rebay S (1997) A high-order accurate discontinuous finite element method for the numerical solution of the compressible Navier–Stokes equations. *Journal of Computational Physics* 131: 267–279.

Burstedde C, Wilcox LC and Ghattas O (2011) `p4est`: Scalablealgorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing* 33: 1103–1133.

Carpenter M and Kennedy C (1994) *Fourth-order 2N-storage Runge–Kutta schemes*. NASA Technical Memorandum 109112. Hampton: NASA.

Chan J and Warburton T (2015) GPU-accelerated Bernstein–Bezier discontinuous Galerkin methods for wave problems. *arXiv:1512.06025*.

Chan J, Wang Z, Modave A, et al. (2015) GPU-accelerated discontinuous Galerkin methods on hybrid meshes. *arXiv:1507.02557*.

Cockburn B and Shu C (1998) The Runge–Kutta discontinuous Galerkin method for conservation laws V: Multidimensional systems. *Journal of Computational Physics* 141: 199–224.

Deville M, Fischer P and Mund E (2002) *High-Order Methods for Incompressible Fluid Flow*. Cambridge: Cambridge University Press.

Fuhry M, Giuliani A and Krivodonova L (2014) Discontinuous Galerkin methods on graphics processing units for

nonlinear hyperbolic conservation laws. *Numerical Methods in Fluids* 76: 982–1003.

Gandham R, Medina D and Warburton T (2014) GPU accelerated discontinuous Galerkin methods for shallow water equations. *arXiv:1403.1661*.

Giraldo FX (1998) The Lagrange–Galerkin spectral element method on unstructured quadrilateral grids. *Journal of Computational Physics* 147: 114–146.

Giraldo FX and Restelli M (2008) A study of spectral element and discontinuous Galerkin methods for the Navier–Stokes equations in nonhydrostatic mesoscale atmospheric modeling: Equation sets and test cases. *Journal of Computational Physics* 227: 3849–3877.

Giraldo FX and Rosmond TE (2004) A scalable spectral element Eulerian atmospheric model (SEE-AM) for NWP: Dynamical core tests. *Monthly Weather Review* 132: 133–153.

Goddeke D, Strzodka R and Turek S (2005, September) Accelerating double precision FEM simulations with GPUs. In: *Proceedings of ASIM eighteenth symposium on simulation technique*, Montreal, Canada, 10–13 October 2005, pp: 139–144.

Kelly JF and Giraldo FX (2012) Continuous and discontinuous Galerkin methods for a scalable three-dimensional nonhydrostatic atmospheric model: Limited area mode. *Journal of Computational Physics* 231: 7988–8008.

Klöckner A (2014) Loo.py: transformation-based code generation for GPUs and CPUs. In: *ArXiv e-prints*. arXiv: 1405.7470 [cs.PL].

Klöckner A, Warburton T, Bridge J, et al (2009) Nodal discontinuous Galerkin methods on graphics processors. *Journal of Computational Physics* 228: 7863–7882.

Marras S, Kelly JF, Moragues M, et al (2016) A review of element-based Galerkin methods for numerical weather prediction: Finite elements, spectral elements, and discontinuous Galerkin. *Archives of Computational Methods in Engineering* 23: 23(4): 673–722. DOI: 10.1007/s11831-015-9152-1.

Medina D, Amik SC and Warburton T (2014) OCCA: A unified approach to multi-threading languages. *arXiv:1403.0968*.

Micikevicius P (2009) 3D finite difference computation on GPUs using CUDA. In: *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-2)*, Washington, DC, USA, 8 March 2009, pp. 79–84. New York, NY, USA: ACM. DOI: 10.1145/1513895.1513905.

Modave A, St-Cyr A and Warburton T (2015) GPU performance analysis of a nodal discontinuous Galerkin method for acoustic and elastic models. *arXiv:1602.07997*.

Müller A, Kopera M, Marras S, et al. (2016) Strong scaling for numerical weather prediction at petascale with the atmospheric model NUMA. *Submitted to: International Journal of High Performance Computing*.

Nair RD, Levy MN and Lauritzen PH (2011) Emerging numerical methods for atmospheric modeling. In: Lauritzen PH, Jablonowski C, Taylor MA, et al. (eds) *Numerical Techniques for Global Atmospheric Models*. Berlin, Heidelberg: Springer, pp. 251–311. DOI: 10.1007/978-3-642-11640-7_9.

Norman M, Larkin J, Vose A, et al. (2015) A case study of CUDA FORTRAN and OpenACC for an atmospheric climate kernel. *Journal of Computational Science* 9: 1–6.

Remacle J, Gandham R and Warburton T (2015) GPU accelerated spectral finite elements on all-hex meshes. *arXiv:1506.05996*

Robert A (1993) Bubble convection experiments with a semi-implicit formulation of the Euler equations. *Journal of Atmospheric Science* 50: 1865–1873.

Sawyer W (2014) An overview of GPU-enabled atmospheric models. In: *ENES workshop on exascale technologies and innovation in HPC for climate models*, DKRZ, Hamburg, 18 March 2014.

Siebenborn M, Schulz V and Schmidt S (2012) A curved-element unstructured discontinuous Galerkin method on GPUs for the Euler equations. *Computing and Visualization in Science* 15: 61–73.

Straka J, Wilhelmson R, Wicker L, et al. (1993) Numerical solutions of a nonlinear density current: A benchmark solution and comparison. *International Journal of Numerical Methods* 17: 1–22.

Tomita H and Satoh M (2005) A new dynamical framework of non hydrostatic global model using the icosahedral grid. *Fluid Dynamics Research* 34: 357–400.

Ullrich P and Jablonowski C (2012) Operator-split Runge–Kutta–Rosenbrock methods for nonhydrostatic atmospheric models. *Monthly Weather Review* 140: 1257–1284.

Volkov V (2010) Better performance at lower occupancy. In: *GPU technology conference (GTC)*, Santa Clara, CA, USA, 25 March 2010.

Williams S, Waterman A and Patterson D (2009) Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM* 52: 65–76.

Yelash L, Müller A, Lukàčová-Medvid'ovà M, et al. (2014) Adaptive discontinuous evolution Galerkin method for dry atmospheric flow. *Journal of Computational Physics* 268: 106–133.

Author biographies

Daniel S Abdi is currently a research associate at the Naval Postgraduate School (NPS), where he works on porting the Non-hydrostatic Unified Model of the Atmosphere (NUMA) to different kinds of manycore processors using a novel programming language that allows one to 'write once and run everywhere'. He received his BS from Addis Ababa University in 2003, his MS from the Indian Institute of Technology in 2006, and his PhD from the University of Western Ontario in 2014, all in Civil Engineering.

Lucas C Wilcox received a BS in Mathematical and Computer Sciences from the Colorado School of Mines in 2001. He graduated with a ScM and PhD in Applied Mathematics from Brown University in 2002 and 2006, respectively. He was an ICES Postdoctoral Fellow at the Institute for Computational Engineering and Science at the University of Texas at Austin, and

is currently an Associate Professor in the Department of Applied Mathematics at the NPS.

Timothy C Warburton received his PhD in Applied Mathematics at Brown University in 1999 and subsequently was a postdoc at the University of Oxford and Brown University. He joined the University of New Mexico as an Assistant Professor in 2001 before moving to Rice University in 2004. In 2015 he took up the John K. Costain Faculty Chair in the College of Science and a professorship in Mathematics at Virginia Tech.Tim has co-authored a popular text on Nodal Discontinuous Galerkin methods as well as numerous journal articles, a third of which are related to high-performance computing, focusing in particular on using graphics processing units to accelerate numerical PDE solvers.

Francis X Giraldo received his BS in Mechanical and Aerospace Engineering from Princeton University in 1987 and his PhD from the University of Virginia in 1995 on 3D adaptive mesh refinement for the Euler equations on high-performance computers. Since then, he has worked primarily in the areas of high-order spatial discretization methods (continuous and discontinuous Galerkin methods), time-integration methods, and high-performance computing. He is the principal architect of NUMA, which will be used as the engine inside the US Navy's next-generation weather prediction system. He is a full professor in the Department of Applied Mathematics at the NPS and a founding member of the Scientific Computing Group.