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

Abdi, Daniel

Sage Publications
A GPU Accelerated Continuous and Discontinuous Galerkin Non-hydrostatic Atmospheric Model

Daniel Abdi¹ and Lucas Wilcox¹ and Timothy Warburton ² and Francis Giraldo¹

Abstract
We present a 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 upto 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 16384 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 (Marras et al. 2015; Giraldo and Rosmond 2004; Kelly and Giraldo 2012; Giraldo and Restelli 2008).

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 which ported the whole dynamical core, a maximum overall speedup of 3 times (from here on, we shall use, e.g., 3x 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 speed of 3x for the dynamical core using CUDA. A comparison of the acceleration of CAM-SE tracer kernels using OpenACC, though substantially easier to program, performed 1.5x slower than the CUDA version (Norman et al. 2015). This could occur, 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.
namely continuous Galerkin (CG) and discontinuous Galerkin (DG), are ported to the GPU in a unified manner (see Sec. 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 speedup the solution of various partial differential equations (PDEs) by orders of magnitude. Acceleration of a CG simulation using GPUs is 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 nontrivial adjustments to the DG method are required to solve nonlinear 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 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 18x over the serial implementation of the method running on a single core CPU. Fuhrty et al. (2014) made an implementation of the 2D discontinuous Galerkin on the GPU using triangular elements and obtained a speedup of about 50x 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 Fuhrty 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 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 Preconditioned 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 (see (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

<table>
<thead>
<tr>
<th>Model</th>
<th>Non-hydrostatic</th>
<th>Method</th>
<th>GPU ported</th>
<th>Speedup</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAM-SE</td>
<td>No</td>
<td>SE</td>
<td>Parts of DyCore</td>
<td>3x</td>
<td>CUDA+OpenACC</td>
</tr>
<tr>
<td>WRF</td>
<td>Yes</td>
<td>FD</td>
<td>Parts of DyCore</td>
<td>2x</td>
<td>CUDA+OpenACC</td>
</tr>
<tr>
<td>NICAM</td>
<td>Yes</td>
<td>FV</td>
<td>DyCore</td>
<td>3x</td>
<td>OpenACC</td>
</tr>
<tr>
<td>ICON</td>
<td>Yes</td>
<td>FV</td>
<td>DyCore</td>
<td>2x</td>
<td>CUDA+OpenACC+OpenCL</td>
</tr>
<tr>
<td>GEOS-5</td>
<td>Yes</td>
<td>FV</td>
<td>Parts of DyCore</td>
<td>5x</td>
<td>CUDA+OpenACC</td>
</tr>
<tr>
<td>FIM/NIM</td>
<td>Yes</td>
<td>FV</td>
<td>DyCore + Physics</td>
<td>3x</td>
<td>F2C-ACC + OpenACC</td>
</tr>
<tr>
<td>GRAPES</td>
<td>Yes</td>
<td>SL</td>
<td>Parts of DyCore</td>
<td>4x</td>
<td>CUDA</td>
</tr>
<tr>
<td>COSMO</td>
<td>Yes</td>
<td>FD</td>
<td>DyCore + Physics</td>
<td>2x</td>
<td>CUDA+OpenACC</td>
</tr>
</tbody>
</table>
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, in (Modave et al. 2015) 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 \((\rho, U, \Theta)\T\), where \(\rho\) is density, \(U = (U, V, W) = \rho u, \Theta = \rho \theta\), where \(\theta\) is potential temperature and \(\mathbf{u} = (u, v, w)\) are the velocity components. We write the governing equations in the following way

\[
\begin{align*}
\frac{\partial \rho}{\partial t} + \nabla \cdot \mathbf{U} &= 0 \\
\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot \left( \frac{\mathbf{U} \otimes \mathbf{U}}{\rho} + P' \mathbf{I}_3 \right) &= -\rho \mathbf{g} \\
\frac{\partial \Theta}{\partial t} + \nabla \cdot \left( \frac{\Theta \mathbf{U}}{\rho} \right) &= 0,
\end{align*}
\]

where \(\mathbf{g}\) is the gravity vector. The pressure in the momentum equation is obtained from the equation of state

\[
P = P_0 \left( \frac{R \Theta}{T_0} \right)^\gamma
\]

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

\[
\begin{align*}
\rho(x, t) &= \overline{\rho}(x) + \rho'(x, t) \\
\Theta(x, t) &= \overline{\Theta}(x) + \Theta'(x, t) \\
P(x, t) &= \overline{P}(x) + P'(x, t),
\end{align*}
\]

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

\[
\begin{align*}
\frac{\partial \rho'}{\partial t} + \nabla \cdot \mathbf{U} &= 0 \\
\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot \left( \frac{\mathbf{U} \otimes \mathbf{U}}{\rho} + P' \mathbf{I}_3 \right) &= -\rho' \mathbf{g} \\
\frac{\partial \Theta'}{\partial t} + \nabla \cdot \left( \frac{\Theta \mathbf{U}}{\rho} \right) &= 0.
\end{align*}
\]

In compact vector notation form

\[
\frac{\partial \mathbf{q}}{\partial t} + \nabla \cdot \mathbf{F}(\mathbf{q}) = \mathbf{S}(\mathbf{q})
\]

where \(\mathbf{q} = (\rho', \mathbf{U}, \Theta')\T\) is the solution vector, \(\mathbf{F}(\mathbf{q}) = (\mathbf{U}, \frac{\mathbf{U} \otimes \mathbf{U}}{\rho} + P' \mathbf{I}_3, \frac{\Theta \mathbf{U}}{\rho})\T\) is the flux vector, and \(\mathbf{S}(\mathbf{q}) = (0, -\rho' \mathbf{g}, 0)\T\) is the source vector.

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

\[
\frac{\partial \mathbf{q}}{\partial t} + \nabla \cdot \mathbf{F}(\mathbf{q}) = \mathbf{S}(\mathbf{q}) + \nabla \cdot (\mu \nabla \mathbf{q})
\]

where \(\mu\) 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 element-based Galerkin (EBG) methods, namely continuous Galerkin and discontinuous Galerkin, is conducted by decomposing the domain \(\Omega \subseteq \mathbb{R}^3\) into \(N_e\) non-overlapping hexahedra elements \(\Omega_e\)

\[
\Omega = \bigcup_{e=1}^{N_e} \Omega_e.
\]

A key property of hexahedral elements is that they allow the use of a tensor product approach thereby decreasing the complexity (in 3D) from \(O(N^6)\) to \(O(N^3)\) 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 (see, e.g., Giraldo (1998)).

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

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

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

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

where \(\{\xi_i\}_{i=1}^{N+1}\) is the set of interpolation points in \([-1, 1]\). In a nodal Galerkin approach, \(\psi_i(\xi)\) are Lagrange polynomials.

*The gravity vector is constant in mesoscale models whereas it varies with location in global scale models.*
associated with a specific set of points; here we choose the Legendre-Gauss-Lobatto (LGL) points \( \{ \xi_i \} \in [-1, 1] \) which are the roots of
\[
(1 - \xi^2)P_N'(\xi)
\]
where \( P_N(\xi) \) is the \( N^{th} \) degree Legendre polynomial. These points are also used for integration with quadrature weights given by
\[
\omega_i = \frac{2}{N(N+1)} \left( \frac{1}{P_N'(\xi_i)} \right)^2.
\]
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
\[
\begin{align*}
\frac{\partial \psi_{ijk}}{\partial \xi} (\xi_a, \eta_b, \zeta_c) &= \frac{d\psi_{ijk}}{d\xi}(\xi_a) \otimes \delta_{bj} \otimes \delta_{ck} \\
\frac{\partial \psi_{ijk}}{\partial \eta} (\xi_a, \eta_b, \zeta_c) &= \delta_{ai} \otimes \frac{d\psi_{ijk}}{d\eta}(\eta_b) \otimes \delta_{ck} \\
\frac{\partial \psi_{ijk}}{\partial \zeta} (\xi_a, \eta_b, \zeta_c) &= \delta_{bj} \otimes \delta_{ai} \otimes \frac{d\psi_{ijk}}{d\zeta}(\zeta_c).
\end{align*}
\]
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 \( \frac{d\omega}{d\chi} \) where \( \chi \) is one of the following: \( \xi, \eta, \zeta \). The derivatives with respect to the physical coordinates \( x = (x, y, z) \) are computed using the Jacobian matrix transformation
\[
\nabla \phi = J \tilde{\nabla} \phi
\]
where \( \tilde{\nabla} \) is the derivative with respect to the reference coordinates \( (\xi, \eta, \zeta) \) and
\[
J = \begin{bmatrix}
\frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} & \frac{\partial x}{\partial \zeta} \\
\frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} & \frac{\partial y}{\partial \zeta} \\
\frac{\partial z}{\partial \xi} & \frac{\partial z}{\partial \eta} & \frac{\partial z}{\partial \zeta}
\end{bmatrix}.
\]

### 3.1 Continuous Galerkin method

Starting from the differential form of the Euler equations in vector notation, shown in Eq. (4), and then expanding with basis functions, multiplying by a test function \( \psi_i \), and integrating yields the element-wise formulation
\[
\int_{\Omega_e} \psi_i \frac{\partial q_N}{\partial t} d\Omega_e + \int_{\Omega_e} \psi_i \nabla \cdot \mathcal{F} d\Omega_e = \int_{\Omega_e} \psi_i S(q_N) d\Omega_e.
\]

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
\[
\int_{\Omega_e} \psi_i \frac{\partial q_N}{\partial t} d\Omega_e + \int_{\Gamma_e} \psi_i \hat{n} \cdot \mathcal{F} d\Gamma_e = \int_{\Omega_e} \nabla \psi_i \cdot \mathcal{F} d\Omega_e = \int_{\Omega_e} \psi_i S(q_N) d\Omega_e
\]
\[
\int_{\Gamma_e} \psi_i \hat{n} \cdot \mathcal{F} d\Gamma_e = \int_{\Omega_e} \psi_i S(q_N) d\Omega_e.
\]
where \( \hat{n} \) is the outward pointing normal on the boundary of the element \( \Gamma_e \). 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., \( F^+ = F^- \). Eqs. (8) and (9) are the strong and weak continuous Galerkin (CG) formulations, respectively, with the finite dimensional space defined as a subset of the Sobolev space
\[
V^CG_N = \{ \psi \in H^1(\Omega) | \psi \in P_N \}
\]
where \( P_N \) defines the set of all \( N \)th degree polynomials. Automatically, \( V^CG_N \subset C^0(\Omega) \), 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
\[
V^DG_N = \{ \psi \in L^2(\Omega) | \psi \in P_N \}.
\]

Therefore \( F^+ \) and \( F^- \) are not equal anymore, hence, we define a numerical flux \( F^* \) as an approximate solution to a Riemann problem to be used in the weak form DG
\[
\int_{\Omega_e} \psi_i \frac{\partial q_N}{\partial t} d\Omega_e + \int_{\Gamma_e} \psi_i \hat{n} \cdot F^* d\Gamma_e = \int_{\Omega_e} \nabla \psi_i \cdot F d\Omega_e = \int_{\Omega_e} \psi_i S(q_N) d\Omega_e
\]
\[
\int_{\Gamma_e} \psi_i \hat{n} \cdot F d\Gamma_e = \int_{\Omega_e} \psi_i S(q_N) d\Omega_e.
\]
where the Rusanov flux, suitable for hyperbolic equations, is defined as
\[
F(q)^* = \{ F(q) \} - \frac{\triangle}{2}[q]
\]
where \( \triangle \) is the speed of sound, \( [\cdot] \) represent an average and \( \{\cdot\} \) represent a jump across a face (from \( \Omega_e \), to its neighbor). If \( C^0 \)-continuity is enforced on the weak form DG in Eq. (10), i.e., \( F = F^* \), it reduces to the weak form CG in Eq. (9).

A strong form DG that resembles Eq. (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 \( \psi_i \) as follows
\[
\int_{\Omega_e} \psi_i \frac{\partial q_N}{\partial t} d\Omega_e + \int_{\Gamma_e} \psi_i \hat{n} \cdot (F^* - F)(q_N) d\Gamma_e + \int_{\Omega_e} \psi_i \nabla \cdot F(q_N) d\Omega_e = \int_{\Omega_e} \psi_i S(q_N) d\Omega_e.
\]
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 Eq. (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 follow 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 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 $\Delta t$ 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 $q$ and its residual $dq$ of size $N$ each, thereby, resulting in a 2N-storage scheme. Given the initial value problem

$$\frac{dq}{dt} = \mathcal{R}(q) \quad \text{with} \quad q(t_0) = q_0$$

the updates at each stage $j$ are conducted as follows

$$\begin{align*}
    dq_j &= A_j dq_{j-1} + \Delta t \mathcal{R}(q_{j-1}) \\
    q_j &= q_{j-1} + B_j dq_j
\end{align*}$$

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 prohibit them from being used in operational settings. NUMA includes Implicit-Explicit (IMEX) methods that allow for much larger time steps, 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 preconditioner, 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.

#### 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

| 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 | -1.602529506 | 1.426311463 |
| 5 | -1.778267193 | 0.197876053 |
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 of 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 AoS 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 \texttt{float3} on the GPU.

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 Sec. 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 polynomials 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.

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 time step 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

\[
\begin{align*}
\int_{\Omega_{e}} \psi \frac{\partial q}{\partial t} \, d\Omega_e + \int_{\Omega_{e}} \psi (\nabla \cdot F - S) \, d\Omega_e + \\
\int_{\Gamma_{e}} \psi \mathbf{n} \cdot (F^e - F) \, d\Gamma_e = \int_{\Omega_{e}} \psi (\nabla \cdot \mu \mathbf{q}) \, d\Omega_e.
\end{align*}
\]

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 Legendre-Gauss-Lobatto (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 five 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 need for a block size that best fits the underlying hardware limits. In traditional GPU kernels, for instance the time step update kernel discussed in Sec. 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 Alg. 1): five prognostic variables \((\rho, U, V, W, \theta)\), pressure \(P\) and two variables for moisture (here, we omit precipitation). The gradient of four variables, which are stored as one \texttt{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 right-hand side vector as shown in Alg. 2.
Figure 1. Volume integral contribution of a horizontal and vertical slice of a 3D element with $4^{th}$ 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-, z- directions, i.e., purple and green nodes on the horizontal slice and light-blue nodes on the vertical slice.

Algorithm 1 GPU algorithms for computing gradient, divergence and Laplacian.

```plaintext
procedure GRADIV(q, grad, div, compute)  
    ▷ Compute gradient or divergence
    Memory fence
    for k,j,i ∈ {0...Nq} do
        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
            qx += sD[i][n] × sq[k][j][n]
            qy += sD[j][n] × sq[k][n][i]
            qz += sD[k][n] × sq[n][j][i]
        if compute = GRAD then
            grad·x = (qx × Irx + qy × Isx + qz × Itx)
            grad·y = (qx × Irj + qy × Iss + qz × Ity)
            grad·z = (qx × Irm + qy × Ism + qz × Itz)
        else if compute = DIVX then
            div = (qx × Irx + qy × Isx + qz × Itx)
        else if compute = DIVY then
            div += (qx × Irj + qy × Iss + qz × Itx)
        else if compute = DIVZ then
            div += (qx × Irm + qy × Ism + qz × Itx)

procedure GRAD(q, grad)  
    call GRADIV(q, grad, -, GRAD)

procedure DIV(q, div)  
    call GRADIV(q·x, -, div, DIVX)
    call GRADIV(q·y, -, div, DIVY)
    call GRADIV(q·z, -, div, DIVZ)

procedure LAP(q, lap)  
    call GRAD(q, gq)
    call DIV(gq, lap)
```

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. 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 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 start up, 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 Fig. 1. Then, we consider three ways of exploiting the shared memory. 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 cache effects, the naïve approach reads $3N_{LGL}$ values from memory to compute the gradient of a variable at a node, for a total of $N_{LGL}^3 \times N_{LGL}^2$ 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 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 $N_{LGL}^3 \times 2$ memory reads.

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 $3^d$ 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 $8 \times (3 + 1)^2 = 512$ threads are launched per multiprocessor. We would like to run with high order polynomial approximations and also have kernels that are efficient for low order polynomial approximations.

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 run-time 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 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 $N_{LGL}^3 \times 2$ memory reads.
block and slice worker, its shared memory consumption will increase in proportion with $N_e \times N_s$, as shown in Alg. 2.

5.3.2 Surface kernel The surface integration, shown in Alg. 3, 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 Fig. 2. A second option is to use hardware atomic operations to update the flux terms. However, hardware support for atomic operations on double precision floating point operations is not universally supported by all GPUs at this time.

5.3.3 Update kernel The time step 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 time stepping is only conditionally stable depending on the Courant number. The implementation of implicit-explicit and fully implicit time stepping 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 time step 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 direct stiffness summation (DSS) operation is implemented in two steps, namely gather and scatter stages. The DSS kernel, shown in Alg. 4, 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 Sec. 6.

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 †. The stabilizing term, shown in Eq. (5), is in divergence form $\nabla \cdot (\mu \nabla q)$ 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 $\mu \nabla^2 q$.

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

$$\int_{\Omega_e} \psi_i \nabla \cdot (\mu \nabla q)d\Omega_e = \int_{\Gamma_e} \psi_i \hat{n} \cdot (\mu \nabla q^+) - \int_{\Omega_e} \nabla \psi_i \cdot (\mu \nabla q)d\Omega_e$$

(13)

and in the strong form as

$$\int_{\Omega_e} \psi_i \nabla \cdot (\mu \nabla q)d\Omega_e = \int_{\Gamma_e} \psi_i \hat{n} \cdot (\mu \nabla q^+ - \mu \nabla q) + \int_{\Omega_e} \psi_i \nabla \cdot (\mu \nabla q)d\Omega_e.$$  

(14)

If we, then, ensure $C^1$-continuity in the CG discretization, i.e. by applying DSS on the gradient so that $\nabla q = \nabla q^+$, 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 out at interior boundaries due to $\nabla q^+ = \nabla q^-$. 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 in Alg. 1. The volume kernel for the the weak form discretization is shown in Alg. 5. The first step in this kernel is to load the field variable $q$ into the fast shared memory. Then, we compute and store the local gradients at each LGL node similar to what is done in the volume kernel. The shared memory requirement of this kernel is rather high due to the need for temporarily storing the gradients besides the field variable. 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

† Hyper-diffusion can also be used but in order to simplify the exposition, we shall only remark on second order diffusion.
Algorithm 3 Surface kernel

map\[3\][2] = ((0,5),(1,3),(2,4))

\textbf{procedure SURFACEKERNEL}(q, R)
  \textbf{for} outerId0 \textbf{do}
    \textbf{for} innerId2 \textbf{do}
      wId = innerId2 mod \(N_s\)
      eId = innerId2 div \(N_s\)
      \textbf{for wId to 2 step} \(N_s\) \textbf{do}
        \textbf{for b=0 to 2 do}
          slId = map[b][wId];
          \textbf{for} j,i \in \{0 \ldots Nq\} \textbf{do}
            e = \(N_k \times \text{outerId0} + \text{eId}\)
            Load face normal \(\hat{n}\) and lift coefficient \(L\)
            Compute maximum wave speed \(|\lambda| = |\hat{n} \cdot u| + \sqrt{|p/p|}\)
            Compute Rusanov flux \(F(q)^* = (\hat{n} \cdot (F(q)^* - F(q)))\)
          R += \(L \times \hat{n} \cdot (F(q)^* - F(q))\)

Algorithm 4 DSS kernel

\textbf{procedure DSSKERNEL}(Q, Qcont, starts, indices, nGlobal, wgt)
  \textbf{for} outerId0 \textbf{do}
    n = outerId0
    \textbf{if} n \leq nGlobal \textbf{then}
      start = starts[n]
      end = starts[n+1]
      \textbf{gQ} = 0
      \textbf{for} m=start \textbf{to} end \textbf{do}
        ind = indices[m]
        \textbf{if} ind \geq 0 \textbf{then}
          pw = wgt[ind];
          \textbf{gQ} += Q[ind] \times pw
          Qcont[n] = gQ
      \textbf{for} m=start \textbf{to} end \textbf{do}
        ind = indices[m]
        \textbf{if} ind \geq 0 \textbf{then}
          Q[ind] = Qcont[n]

processing, can be used here as well. After computing the local gradients, the \(\nabla v_i \cdot \mu \nabla q_j\) 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 \(q\) for which we already applied DSS at the end of the time step or RK-stage, and \(\nabla q\), for which we ignore DSS for efficiency reasons discussed later in this section. In case of hyper-viscosity of
order 3 or more, the DSS on $\nabla q$ maybe required to ensure at least $C^1$ continuity.

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

$$\nabla q = Q$$
$$\nabla \cdot (\mu \nabla q) = \nabla \cdot (\mu Q)$$

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

$$\int_{\Omega_e} \psi_i Q d\Omega_e = \int_{\Gamma_e} \psi_i \hat{n} \cdot (q^* - q) d\Gamma_e + \int_{\Omega_e} \psi_i \nabla q d\Omega_e .$$

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

$$\int_{\Omega_e} \psi_i \nabla \cdot (\mu \nabla q) d\Omega_e = \int_{\Gamma_e} \psi_i \hat{n} \cdot (\mu Q^* - \mu Q) d\Gamma_e + \int_{\Omega_e} \psi_i \nabla \cdot (\mu Q) d\Omega_e .$$

According to (Bassi and Rebay 1997), we use centered fluxes for both $q$ and $Q$ such that $q^* = \{q\}$ and $Q^* = \{Q\}$. The mixed form is implemented directly by first computing the volume integral of the gradient in Eq. (16) using Alg. 1, and then modifying the result with the surface integral contribution computed using centered fluxes $q^* = \{q\}$. 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 Eq. (17) are added to the right-hand side 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 numerical weather prediction (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 up to tens of thousands of CPUs was demonstrated. More recently the scalability of the implementation is tested on the Mira supercomputer, located at Argonne National Laboratory, using 3.1 million MPI ranks (Müller et al. 2016). NUMA achieved linear scalability for both explicit and 1D implicit-explicit (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 start up. 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 sub-domains in the CG spatial discretization is achieved by the Direct
Algorithm 5 Laplacian diffusion kernel

\begin{algorithm}
\KwProcedure{LAPLACE}{$Q, rhs, nu$}
\textbf{Shared} sq, sqr, sqs, sqt all arrays of size of [Nq][Nq][Nq] \\textbf{Memory fence}
\For{k,j,i \in \{0 \ldots Nq\}}{
    sq[k][j][i] = q
}
\textbf{Memory fence}
\For{k,j,i \in \{0 \ldots Nq\}}{
    qr += sD[i][n][sq[k][j][n]];
    qs += sD[j][n][sq[k][n][i]];
    qt += sD[k][n][sq[n][j][i]];
    sqr[k][j][i] = \mu(G11 \times qr + G12 \times qs + G13 \times qt);
    sqs[k][j][i] = \mu(G12 \times qr + G22 \times qs + G23 \times qt);
    sqt[k][j][i] = \mu(G13 \times qr + G23 \times qs + G33 \times qt);
}
\textbf{Memory fence}
\For{k,j,i \in \{0 \ldots Nq\}}{
    lapq = 0
}
\For{n \in \{0 \ldots Nq\}}{
    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 \times lapq
}\end{algorithm}

Stiffness Summation (DSS) operator which imposes $C^0$ continuity of solutions at element interfaces. The DSS operator is applied both to the mass matrix and the right-hand side (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. Alg. 6 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 sub-domains 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. Alg. 7 shows an outline of a multi-GPU DG implementation with communication-computation overlap. The latest technology in GPUs allow 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 $\nabla q$ for the LDG stabilization method is also done similarly.

7 Performance tests

7.1 Speedup results

First, we present speedup results for the GPU implementation of NUMA against the base Fortran code. 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 on different GPU cards. We will present the details of the test cases later in Sec. 8; here we give the workload of each problem:

1. 2D Rising-thermal bubble: 100 elements with polynomial order 7, for a total of 51200 nodes

The base Fortran code is the original CPU code, i.e., the non-OCCA implementation that we use on the GPUs.
Algorithm 6 DSS on the GPU for the RHS vector

```plaintext
procedure DSS(RHS)
    Gather RHS
    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
        Boundary data is used to update the RHS vector
    Scatter RHS

   ▷ See Alg. 4 for details
```

2. 3D Rising-thermal bubble: 1000 elements with polynomial order 5, for a total of 216000 nodes
3. Acoustic wave on the sphere: 1800 elements with polynomial order 4, for a total of 225000 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 2.2GHz AMD CPU. The specs for the GPU cards, bandwidth and double precision TFLOPS/s, are as follows:
- C2070: 144 GB/s, 0.5 TFLOPS/s
- Titan black: 336 GB/s, 1.7 TFLOPS/s
- and K20X: 225 GB/s, 1.3 TFLOPS/s.

Next, we present performance tests on the Titan supercomputer located at the Oak Ridge National Laboratory, where each node has a K20X GPU card and an AMD Opteron 6274 CPU with 16 cores at 2.2 GHz. The GPU card has 2,688 cores at 0.732 GHz, 6 GB memory, 250 GB/s bandwidth with peak performances of 1.31 and 3.95 teraflops in double and single precision, respectively. The speedup results are reported relative to the NUMA Fortran code using all 16 cores of the CPU. We will examine the different kernel design and parameter choices we made in Sec. 5 using the 2D rising thermal bubble benchmark problem. The problem size is increased progressively from 10x10=100 elements until we fill up all the memory available on the device at 160x160=25600 elements. The first test result, presented in Table 4, evaluates the 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 the good performance at polynomial order 7 is due to the thread block sizes 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 single precision (SP) arithmetic than double precision (DP). For instance, the SP peak performance of a K20X GPU is 3x more than its DP peak performance. In Table 5, we present the speedup results comparing SP and DP performance. We obtain a maximum speedup of about 15x and 11x using single and double precision calculations, respectively. The reason for different speedup numbers for SP and DP is that NUMA running on the CPU is able to achieve a speedup of only 1.5x 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 6 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 5; the block size, when processing two elements per block, exceeds the hardware limit at polynomial orders above 5. 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 4 — for which performance remains more or less the same. We mentioned earlier that polynomial order 4 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 Sec. 5 that using vector datatype \texttt{float4} to store field variables may help to improve performance because one load operation is issued when fetching a \texttt{float4} data instead of four. Table 7 compares the speedup obtained using \texttt{float1} and \texttt{float4} versions of the volume kernel. The \texttt{float4} version performs better in most of the cases; here, again, the performance at polynomial order 4 is more or less the same.

We discussed in Sec. 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 tested so far up to polynomial order 7. First, we compare the performance of the four ways to use shared and L1 cache memory; namely, the naive, Shared-1, Shared-2 and two-pass (horizontal+vertical) methods. Fig. 3 shows that the two-pass method performs the best — about two times better than the naive approach that does not use shared 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

**Table 3.** Speedup comparison between CPU and GPU for both single precision and double precision 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. Two orders of magnitude performance improvement is obtained relative to a single core CPU with the newer cards.

<table>
<thead>
<tr>
<th>Test case</th>
<th>Double precision</th>
<th>Single precision</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU GPU Speedup</td>
<td>CPU GPU Speedup</td>
</tr>
<tr>
<td>2D mb</td>
<td>930.1 27.8</td>
<td>33.4 612.3</td>
</tr>
<tr>
<td>3D mb</td>
<td>4408.9 141.9</td>
<td>31.1 3970.0</td>
</tr>
<tr>
<td>Acoustic wave</td>
<td>3438.8 96.7</td>
<td>35.6 2379.9</td>
</tr>
</tbody>
</table>

**Table 4.** OpenCL vs CUDA: Speedup comparison between CPU and GPU for double precision calculations at different number of elements and polynomial orders using OpenCL and CUDA translation of the native OCCA kernel code. The CPU card is K20X and the GPU is a 16-core 2.2GHz AMD Opteron 6274. The timing (in sec) 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.

<table>
<thead>
<tr>
<th>N</th>
<th>CPU GPU Speedup</th>
<th>CPU GPU Speedup</th>
<th>CPU GPU Speedup</th>
<th>CPU GPU Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>8x8x20=6400 elements</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.46 0.59/0.52</td>
<td>2.37/2.81</td>
<td>10.62</td>
<td>2.57/2.17</td>
</tr>
<tr>
<td>3</td>
<td>2.68 0.60/0.59</td>
<td>3.59/3.14</td>
<td>22.01</td>
<td>6.18/4.19</td>
</tr>
<tr>
<td>4</td>
<td>5.03 0.97/0.86</td>
<td>5.46/6.16</td>
<td>36.03</td>
<td>8.45/6.07</td>
</tr>
<tr>
<td>5</td>
<td>8.12 1.47/1.37</td>
<td>5.52/5.93</td>
<td>77.03</td>
<td>7.32/7.80</td>
</tr>
<tr>
<td>6</td>
<td>13.89 2.27/2.11</td>
<td>6.11/5.68</td>
<td>122.27</td>
<td>7.06/7.59</td>
</tr>
<tr>
<td>7</td>
<td>20.49 2.68/2.41</td>
<td>7.66/6.50</td>
<td>195.61</td>
<td>9.40/9.17</td>
</tr>
</tbody>
</table>

**Table 5.** Double vs Single Precision: Speedup comparison between CPU and GPU for single and double precision calculations at different number of elements and polynomial orders using CUDA translation of OCCA kernel code. A maximum speedup of about 15x is observed. The CPU/GPU times and Speedups are given first for double precision and then for single precision. The GPU card is K20X and the CPU is a 16-core 2.2GHz AMD Opteron 6274.

<table>
<thead>
<tr>
<th>N</th>
<th>CPU GPU Speedup</th>
<th>CPU GPU Speedup</th>
<th>CPU GPU Speedup</th>
<th>CPU GPU Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>10x10=100 elements</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.46 0.59/0.52</td>
<td>2.37/2.81</td>
<td>10.62</td>
<td>2.57/2.17</td>
</tr>
<tr>
<td>3</td>
<td>2.68 0.60/0.59</td>
<td>3.59/3.14</td>
<td>22.01</td>
<td>6.18/4.19</td>
</tr>
<tr>
<td>4</td>
<td>5.03 0.97/0.86</td>
<td>5.46/6.16</td>
<td>36.03</td>
<td>8.45/6.07</td>
</tr>
<tr>
<td>5</td>
<td>8.12 1.47/1.37</td>
<td>5.52/5.93</td>
<td>77.03</td>
<td>7.32/7.80</td>
</tr>
<tr>
<td>6</td>
<td>13.89 2.27/2.11</td>
<td>6.11/5.68</td>
<td>122.27</td>
<td>7.06/7.59</td>
</tr>
<tr>
<td>7</td>
<td>20.49 2.68/2.41</td>
<td>7.66/6.50</td>
<td>195.61</td>
<td>9.40/9.17</td>
</tr>
</tbody>
</table>
Table 6. 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 1 element-per-block and then for the 2 elements-per-block approaches. Improvement in performance is observed using 2 elements-per-block in all the cases except for the 10x10 elements case, which does not fully occupy the GPU device when processing 2-elements-per-block. The GPU card is K20X and the CPU is a 16-core 2.2GHz AMD Opteron 6274.

<table>
<thead>
<tr>
<th></th>
<th>10x10=100 elements</th>
<th>30x30=900 elements</th>
<th>40x40=1600 elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>CPU</td>
<td>GPU</td>
<td>Speedup</td>
</tr>
<tr>
<td>2</td>
<td>1.46</td>
<td>0.52</td>
<td>2.81</td>
</tr>
<tr>
<td>3</td>
<td>2.68</td>
<td>0.59</td>
<td>4.54</td>
</tr>
<tr>
<td>4</td>
<td>5.30</td>
<td>0.86</td>
<td>6.16</td>
</tr>
<tr>
<td>5</td>
<td>8.12</td>
<td>1.37</td>
<td>9.39</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>80x80=6400 elements</th>
<th>120x120=14400 elements</th>
<th>160x160=25600 elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>CPU</td>
<td>GPU</td>
<td>Speedup</td>
</tr>
<tr>
<td>2</td>
<td>80.72</td>
<td>13.33</td>
<td>10.62</td>
</tr>
<tr>
<td>3</td>
<td>2.68</td>
<td>0.59</td>
<td>4.54</td>
</tr>
<tr>
<td>4</td>
<td>5.30</td>
<td>0.86</td>
<td>6.16</td>
</tr>
<tr>
<td>5</td>
<td>8.12</td>
<td>1.37</td>
<td>9.39</td>
</tr>
</tbody>
</table>

Table 7. float4 vs float4: The effect of using float4 for computing the gradient in the volume kernel is compared against the version where one field variable is loaded. The CPU/GPU time and Speedups are given first for float4 and then for float4. Some improvement is observed in most cases except when using polynomial order 4, which results in a good thread block size. The GPU card is K20X and the CPU is a 16-core 2.2GHz AMD Opteron 6274.

<table>
<thead>
<tr>
<th></th>
<th>10x10=100 elements</th>
<th>30x30=900 elements</th>
<th>40x40=1600 elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>CPU</td>
<td>GPU</td>
<td>Speedup</td>
</tr>
<tr>
<td>2</td>
<td>1.46</td>
<td>0.52</td>
<td>2.81</td>
</tr>
<tr>
<td>3</td>
<td>2.68</td>
<td>0.59</td>
<td>4.54</td>
</tr>
<tr>
<td>4</td>
<td>5.30</td>
<td>0.86</td>
<td>6.16</td>
</tr>
<tr>
<td>5</td>
<td>8.12</td>
<td>1.37</td>
<td>9.39</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>80x80=6400 elements</th>
<th>120x120=14400 elements</th>
<th>160x160=25600 elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>CPU</td>
<td>GPU</td>
<td>Speedup</td>
</tr>
<tr>
<td>2</td>
<td>80.72</td>
<td>13.33</td>
<td>10.62</td>
</tr>
<tr>
<td>3</td>
<td>2.68</td>
<td>0.59</td>
<td>4.54</td>
</tr>
<tr>
<td>4</td>
<td>5.30</td>
<td>0.86</td>
<td>6.16</td>
</tr>
<tr>
<td>5</td>
<td>8.12</td>
<td>1.37</td>
<td>9.39</td>
</tr>
</tbody>
</table>

Table 8. Higher order polynomials: The performance of the two pass method, with horizontal + vertical split, is evaluated at higher polynomial orders in double precision 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 2.2GHz AMD Opteron 6274.

<table>
<thead>
<tr>
<th></th>
<th>10x10=100 elements</th>
<th>30x30=900 elements</th>
<th>40x40=1600 elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>CPU</td>
<td>GPU</td>
<td>Speedup</td>
</tr>
<tr>
<td>8</td>
<td>31.17</td>
<td>4.77</td>
<td>6.53</td>
</tr>
<tr>
<td>9</td>
<td>43.21</td>
<td>5.90</td>
<td>7.32</td>
</tr>
<tr>
<td>10</td>
<td>59.89</td>
<td>7.14</td>
<td>8.38</td>
</tr>
<tr>
<td>11</td>
<td>79.86</td>
<td>9.61</td>
<td>9.31</td>
</tr>
<tr>
<td>12</td>
<td>103.64</td>
<td>13.40</td>
<td>7.77</td>
</tr>
<tr>
<td>14</td>
<td>169.49</td>
<td>23.52</td>
<td>7.20</td>
</tr>
<tr>
<td>15</td>
<td>220.91</td>
<td>28.36</td>
<td>7.19</td>
</tr>
</tbody>
</table>

Figure 3. Comparison of different ways of exploiting fast L1 cache and Shared memory in volume kernel. The Speedups are reported relative to the naive approach. The two-pass method performs the best due to better use of shared memory.

in registers, most of it spills to global thread private memory. Because the polynomial order is high and we are loading all field data (8 floats) to registers, the register pressure is too high for the method to show any benefit.
In Table 8, we present the performance of the high-order volume kernel that uses the two-pass method for polynomial orders of 8 to 15. It is not possible to solve bigger size problems than 40x40 elements with polynomial order 15 on this GPU because of the limited memory of 6GB per card. We get a maximum speedup of about 9x at higher order polynomials, which is slightly less than the 11x 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 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) that sets an upper bound on kernel performance based on peak GFLOPS/s and GB/s of the device. We use two approaches to 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 Figs. 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 projection kernels. These tests are conducted on the isentropic vortex problem (see Sec. 8.1), which concerns advection of a vortex by a constant velocity. The GPU is a Tesla K20c GPU with the following specification: 2,496 cores at 0.706 GHz, 5GB memory, 208 GB/s bandwidth with peak performances of 1.17 teraflops and 3.52 teraflops in double and single precision, respectively.

The highest GFLOPS/s observed in any of the kernels is about 320 GFLOPS/s for the horizontal volume kernel at \( N = 10 \) using single precision arithmetic. The vertical volume kernel is a close second, but the surface and update kernels lag far behind in terms of GFLOPS/s performed. The update kernel, which does the explicit Runge-Kutta time integration, shows the highest bandwidth performance at about 208GB/s, which is in fact the peak memory bandwidth of the device. The projection 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 Fig. 5a-5d, are conducted using a GTX Titan Black GPU. For these tests we used mprof, to determine the effective arithmetic throughput and memory bandwidth. As a result, the plots obtained from this test are less smoother than the previous tests which were produced by hand-counting 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 is 700 GFLOPS/s for the volume kernel using single precision. 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 8. The diffusion kernels, used for computing the Laplacian, also show similar performance characteristics as the volume kernels.

7.3 Scalability test

The scalability of the multi-GPU implementation is tested on a GPU cluster, namely, the Titan supercomputer which has 18688 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 Fig. 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 Fig. 7. The scalability increases by more than 20% going from a 100 to 900 elements grid per GPU.

In operational numerical weather prediction (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 Fig. 7, on a global scale simulation problem described in Sec. 8.5. Our goal here is to determine the number of GPUs required for a given simulation time limit for two resolutions: a coarse grid of 13km resolution and a fine grid of 3km resolution.
Prepared using sagej.cls

The grids are cubed sphere with 6x112x112x4 elements and N=7 for the 13km resolution test, and 6x144x144x4 elements and N=7 for the 3km resolution test The plot shows that about 1500 and 8192 GPUs are required to bring down

On cubed sphere grids, the total number of elements are denoted as $N_{panels} \times N_{z} \times N_{x} \times N_{y}$ where $N_{panels} = 6$ for the six panels of the cubed sphere, $N_{z} = N_{y}$ are the number of elements in both horizontal
Figure 6. Scalability test of multi-GPU implementation of NUMA: The scalability of NUMA for up to 16384 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 is conducted using a unified implementation of CG and DG. The efficiency of DG is 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. (left) Scalability test of Multi-GPU implementation for different number of elements using up to 64 nodes of Titan. The 60x60 element grid provides a much better scalability than the 10x10 grid, hence, we expect better scaling results with bigger size problems. (right) Strong scalability test for 3km and 13km resolution global simulation on the sphere.

the simulation time below 100 min for the coarse and fine grids respectively. We believe that once we port the implicit-explicit (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 minute wall clock 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 is validated using a suite of benchmark problems showcasing various characteristics of atmospheric dynamics. We consider problems of different scale: cloud-resolving (micro-scale), limited area (meso-scale) 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 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 non-linearity in the Euler equations.

The free stream conditions are

\[
\rho = 1, \quad u = U_{\infty}, \quad v = V_{\infty}, \quad \theta = \theta_{\infty},
\]

directions on each panel, and \( N_z \) are the number of elements in the vertical direction.
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 \([-5m, 5m] \times [-5m, 5m] \times [-0.5m, 0.5m]\) 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 x 22 x 2 elements with polynomial order of \(N = 7\) in all directions for a total of about 0.5 million nodes. The simulation is run for 10s with a constant time step of \(\Delta t = 0.001s\) using the modified Runge-Kutta time integration scheme discussed in Sec. 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 Fig. 8.

To evaluate the accuracy of the numerical model, we compute the \(L^2\) norm of the error \(q - q^\infty\) over the domain \(\Omega\), i.e., \(||q - q^\infty||_{L^2(\Omega)}\), for both single precision (SP) and double precision (DP) arithmetic, where \(q^\infty\) is the exact solution. The DP run takes about 267s to complete while the SP run takes 161s; however, the maximum error associated with the SP calculations is much larger as shown in Fig. 8e. Therefore, if this reduction in accuracy is acceptable for a certain application, then using single precision 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 Fig. 8c; 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.

### 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 \(\theta_0\). 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

\[
p = p_0 \left(1 - \frac{gz}{c_p \theta_0}\right)^{\gamma / \gamma - 1}.
\]

The potential temperature perturbation is given by

\[
\theta' = \begin{cases} 
0 & \text{for } r > r_c \\
\theta_c (1 + \cos(\pi r^3 / r_c^3)) & \text{for } r \leq r_c 
\end{cases}
\]

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

The simulation is run for 1000s using the explicit Runge-Kutta time integration method discussed in Sec. 4 with a constant time step of \(\Delta t = 0.02s\). The status of the bubble at different times is shown in Fig. 9. The results agree with that reported in (Giraldo and Restelli 2008). Most importantly, the results are identical with that 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 upto 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 right-hand side. This destroys the symmetry of the rising bubble. We should note here that the rising thermal bubble problem in Sec. 8.2 could have been solved considering only half of the domain because of symmetry, which is not the case here. Also, the potential temperature perturbation \(\theta'\) is specified differently for this problem. Within a certain radius \(r_c\), the perturbation is a constant \(\theta_c\); outside of this inner domain, it is defined by a Gaussian profile as

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

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

The result of the simulation is shown in Fig. 10 which confirms the fact that the rising bubble indeed loses its symmetry. The edge of the rising bubble becomes sharper in some places from 600s onwards. Qualitative comparison with the results shown in (Robert 1993; 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.
Figure 8. Isentropic vortex: Plot of density ($\rho$) of the vortex at different times show 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 plot 8d. A grid of 22x22x2 elements with $7^{th}$ degree polynomial is used.

Figure 9. Potential temperature perturbation $\theta'$ (K) contour plot for the 2D rising thermal bubble problem run with CG and an artificial viscosity of $\mu = 1.5 \text{ m}^2/\text{s}$ for stabilization. Results are shown at t=0, 300, 500, 600, 700 and 900 seconds. A grid of 10x1x10 elements with $6^{th}$ degree polynomials is used.

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 $\theta_0$. 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 $\mu = 75 \text{ m}^2/\text{s}$ according to (Straka et al. 1993).

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

Fig. 11 shows the evolution of potential temperature of the bubble up to 900 seconds. The vortical structures formed at t=900 sec, namely three Kelvin-Helmholtz instability rotors, are similar to that shown in (Straka et al. 1993; Ullrich and Jablonowski 2012). The first rotor is formed near the leading edge of the density current at 300 sec, then the second rotor develops at the front of the density current around 600 sec. 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 $\theta_0=300K$. A perturbation pressure $P'$ is superimposed on the reference pressure

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

where

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

$$ g(r) = \sin \left( \frac{r_\psi \phi r}{r_T} \right) $$

where $\Delta P = 100$ Pa, $n_\psi = 1$, $r_c = r_e/3$ 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

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

where $(\lambda_0, \phi_0)$ is the origin of the acoustic wave.

The grid is a cubed sphere $6 \times 10 \times 10 \times 3$ 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 Fig. 12, 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 hours. 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 that is close to the calculated sound speed (a relative error of less than 1%).

9 Conclusions

In this work, we have ported the Non-hydrostatic Unified Model of the Atmosphere (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 15x and 11x for the GPU relative to the CPU using single and double precision 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 16384 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 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 are 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 implicit-explicit (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 time step.

10 Acknowledgement

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 under Contract No. DE-AC05-00OR22725. The authors gratefully acknowledge support from the Office of Naval Research through PE-0602435N.

References


Klöckner A and Warburton T (2013) A loop generation tool for CPUs and GPUs, part i: Data models, algorithms, and heuristics.


