20th AIAA Computational Fluid Dynamics Conference
27 - 30 June 2011, Honolulu, Hawaii

AIAA 2011-3229

On the Development of a High-Order, Multi-GPU Enabled, Compressible Viscous Flow Solver for Mixed Unstructured Grids

Patrice Castonguay*, David M. Williams†, Peter E. Vincent‡, Manuel Lopez§, Antony Jameson¶
Department of Aeronautics and Astronautics, Stanford University, Stanford, CA, 94305

This work discusses the development of a three-dimensional, high-order, compressible viscous flow solver for mixed unstructured grids that can run on multiple GPUs. The solver utilizes a range of so-called Vincent-Castonguay-Jameson-Huynh (VCJH) flux reconstruction schemes in both tensor-product and simplex elements. Such schemes are linearly stable for all orders of accuracy and encompass several well known high-order methods as special cases. Because of the high arithmetic intensity associated with VCJH schemes and their element-local nature, they are well suited for GPUs. The single-GPU solver developed in this work achieves speed-ups of up to 45x relative to a serial computation on a current generation CPU. Additionally, the multi-GPU solver scales well, and when running on 32 GPUs achieves a sustained performance of 2.8 Teraflops (double precision) for 6th-order accurate simulations with tetrahedral elements. In this paper, the techniques used to achieve this level of performance are discussed and a performance analysis is presented. To the authors' knowledge, the aforementioned flow solver is the first high-order, three-dimensional, compressible Navier-Stokes solver for mixed unstructured grids that can run on multiple GPUs.

I. Introduction

In recent years, the development of high-order methods for unstructured grids has been a subject of ongoing research. High-order methods could potentially yield better accuracy and reduced computational costs when compared to low-order methods (order of accuracy ≤ 2) for problems with low error tolerances. However, existing high-order methods for unstructured grids are generally less robust and more complex to implement than their low-order counterparts and these issues must be addressed for their wide-spread use. One of the most promising classes of schemes are the Discontinuous Galerkin (DG) methods, which were originally proposed by Reed and Hill[1] in 1973 to solve the neutron transport problem and for which the theoretical basis has been provided in a series of paper by Cockburn and Shu.[2–5] Among variants of DG methods, the nodal DG approach, for which an exposition can be found in the recent textbook by Hesthaven and Warburton,[6] is perhaps the simplest and most efficient. Similar to the nodal DG method is the spectral difference (SD) method, for which the foundation was first put forward by Kopriva, and Kolias[7]under the name of "stagered grid Chebyshev multidomain" methods. In 2006, Liu, Wang and Vinokur[8] presented a more general formulation for both triangular and quadrilateral elements. In recent years, a range of studies have succesfully employed the SD method to solve a wide range of problems.[9–11] In 2007, the relationship between the nodal DG and the SD methods was clarified by Huynh, who presented the flux reconstruction (FR) approach in a notable paper.[12] Huynh's formulation allows the recovery of well-known high-order schemes, including a particular nodal DG method and the SD method (at least for linear advection). In

*Ph.D. Candidate, Department of Aeronautics and Astronautics, Stanford University, AIAA Student Member
†Ph.D. Candidate, Department of Aeronautics and Astronautics, Stanford University, AIAA Student Member
‡Postdoctoral Scholar, Department of Aeronautics and Astronautics, Stanford University
§Master's Student, Department of Aeronautics and Astronautics, Stanford University
¶Thomas V. Jones Professor of Engineering, Department of Aeronautics and Astronautics, Stanford University, AIAA Member

2009, Huynh[13] extended the FR approach to diffusion problems. Recently, by extending the proof of stability of a SD scheme by Jameson,[14] the authors have identified a range of FR schemes that are guaranteed to be stable (for linear advection) for all orders of accuracy. The 1D formulation of those schemes was presented by Vincent et al.[15] in 2010 and the formulation for triangles was presented by Castonguay et al.[16] in 2011. These energy stable FR schemes are the ones used in the present work and will henceforth be referred to as VCJH (Vincent-Castonguay-Jameson-Huynh) schemes.

The advent of General Purpose Graphical Processing Units (GPGPUs) in the last few years has brought about an important change in the world of computing. GPGPUs, such as the NVIDIA Tesla C2070, are massively parallel processing units that are tailored towards general computing. In recent years, several groups have demonstrated how GPGPUs can be used to dramatically improve the performance of certain algorithms. For example, in 2009, Klockner et al.[17] demonstrated a substantial performance gain when using GPUs to solve the Maxwell's equations using the DG method. Because the FR and DG approaches are similar in many regards, the task of developing a multi-GPU, high-order, unstructured Navier-Stokes solver that uses the FR approach has been undertaken by the authors in the hope of achieving similar levels of performance. The FR method is well suited to GPUs for two main reasons. Firstly, the vast majority of operations are performed in an element-local fashion, without coupling between elements. This locality in memory access allows the efficient use of a fast access memory available on the GPU called shared memory. Secondly, high-order methods typically require more work per degree of freedom than low-order methods, hence they can benefit the most from the high computational throughput of GPUs.

In this work, a multi-GPU enabled, high-order, compressible Navier-Stokes solver that can run on mixed unstructured grids is developed. This paper starts by briefly presenting the FR approach in 1D, on quadrilateral elements, on simplex elements and on prismatic elements. Then, the implementation of the single-GPU and multi-GPUs algorithms is explained and the performance of the algorithms is analyzed. Finally, two realistic test cases are considered, demonstrating the performance of the solver.

II. Flux Reconstruction Method

In this section, a brief overview of the FR approach and VCJH schemes is given. For more details, the reader is encouraged to read the articles by Huynh,[12,13] Vincent et al.,[15] Jameson et al.[18] and Castonguay et al.[16]

A. FR Approach in One Dimension

In this subsection, a review of the FR approach in one dimension is presented. Consider solving the following 1D scalar conservation law

$$\frac{\partial u}{\partial t} + \frac{\partial f}{\partial x} = 0 \tag{1}$$

within an arbitrary domain $\mathbf{\Omega}$, where x is a spatial coordinate, t is time, $u = u(x,t)$ is a conserved scalar quantity and $f = f(u, \frac{\partial u}{\partial x})$ is the flux in the x direction. This scalar equation is a good model for the Navier-Stokes equations. Furthermore, consider partitioning $\mathbf{\Omega}$ into N non-overlapping, conforming elements each denoted $\mathbf{\Omega}_n = \{x | x_n < x < x_{n+1}\}$ such that

$$\mathbf{\Omega} = \bigcup_{n=1}^{N} \mathbf{\Omega}_n. \tag{2}$$

Finally, having partitioned $\mathbf{\Omega}$ into separate elements, consider representing the exact solution u within each $\mathbf{\Omega}_n$ by a function denoted by $u_n^\delta = u_n^\delta(x,t)$ which is a polynomial of degree p within $\mathbf{\Omega}_n$ and zero outside the element. Similarly, consider representing the exact flux f within each $\mathbf{\Omega}_n$ by a function denoted $f_n^\delta = f_n^\delta(x,t)$ which is a polynomial of degree $p+1$ inside $\mathbf{\Omega}_n$ and identically zero outside the element. Thus, the total approximate solution $u^\delta = u^\delta(x,t)$ and the total approximate flux $f^\delta = f^\delta(x,t)$ over the domain $\mathbf{\Omega}$ can be written as

$$u^\delta = \sum_{n=1}^{N} u_n^\delta \approx u, \qquad f^\delta = \sum_{n=1}^{N} f_n^\delta \approx f.$$

In order to simplify the implementation, it is advantageous to transform each $\mathbf{\Omega}_n$ to a standard element $\mathbf{\Omega}_S = \{\xi | -1 \leq \xi \leq 1\}$ via the mapping $\Theta_n(\xi)$,

$$x = \Theta_n(\xi) = \left(\frac{1-\xi}{2}\right) x_n + \left(\frac{1+\xi}{2}\right) x_{n+1}. \tag{3}$$

Having performed such a transformation, the evolution of u_n^δ within any individual $\mathbf{\Omega}_n$ (and thus the evolution of u^δ within $\mathbf{\Omega}$) can be determined by solving the following transformed equation within the standard element $\mathbf{\Omega}_S$

$$\frac{\partial \hat{u}^\delta}{\partial t} + \frac{\partial \hat{f}^\delta}{\partial \xi} = 0, \tag{4}$$

where

$$\hat{u}^\delta = \hat{u}^\delta(\xi, t) = J_n u_n^\delta(\Theta_n(\xi), t) \tag{5}$$

is a polynomial of degree p,

$$\hat{f}^\delta = \hat{f}^\delta(\xi, t) = f_n^\delta(\Theta_n(\xi), t), \tag{6}$$

is a polynomial of degree $p+1$, and $J_n = (x_{n+1} - x_n)/2$.

The FR approach to solving equation (4) within the standard element $\mathbf{\Omega}_S$ consists of seven main stages. The first stage is to define a specific form for $\hat{u}^\delta$. In the FR approach, a set of $N_s^{1D} = p+1$ solution points are defined within $\mathbf{\Omega}_S$, with each point located at a distinct position ξ_i ($i = 1$ to N_s^{1D}). As suggested by the authors,[18,19] the solution points should be located at good quadrature points to minimize aliasing errors. Hence, in 1D, the solution points are located at Gauss points. Two flux points are also defined, which lie at the boundary of the element, as shown in figure (1). The approximate solution $\hat{u}^\delta$ is represented by a polynomial of degree p of the form

$$\hat{u}^\delta = \sum_{i=1}^{N_s^{1D}} \hat{u}_i^\delta \, l_i, \tag{7}$$

where $l_i = l_i(\xi)$ is the 1D Lagrange polynomial associated with solution point i and $\hat{u}_i^\delta = \hat{u}_i^\delta(t)$ are the values of $\hat{u}^\delta$ at the solution points ξ_i.

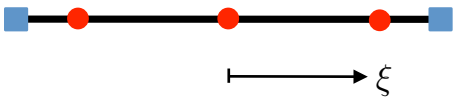

Figure 1: 1D reference element ($\mathbf{\Omega}_S$) for p=2. Solution points are represented by red circles and flux points by blue squares

The second stage of the FR approach involves calculating a common solution value at either end of the standard element $\mathbf{\Omega}_S$ (at $\xi = \pm 1$). In order to calculate this common value, one must first obtain values for the approximate solution at either end of the standard element via equation (7). Once these values have been obtained, they can be used in conjunction with analogous information from adjoining elements to calculate a common interface solution. In what follows, the transformed common interface solution associated with the left and right ends of $\mathbf{\Omega}_n$ will be denoted $\hat{u}_L^{\delta I}$ and $\hat{u}_R^{\delta I}$, respectively. In this work, the BR2 approach of Bassi and Rebay[20] is used to deal with viscous terms, and therefore the common interface solution $u^{\delta I}$ is computed as the average of the solutions from the left and right side of the interface (i.e. $u^{\delta I} = \frac{u^{\delta +} + u^{\delta -}}{2}$ where the superscripts - and + refer to information on the left and right sides of the interface, respectively).

The third stage involves computing a corrected solution gradient, denoted by $\hat{q}^\delta$ which approximates the solution gradient within the reference element. In order to define $\hat{q}^\delta$, consider first defining degree $p+1$ correction functions $g_L = g_L(\xi)$ and $g_R = g_R(\xi)$ that approximate zero (in some sense) within $\mathbf{\Omega}_S$, as well as satisfying

$$g_L(-1) = 1, \qquad g_L(1) = 0, \tag{8}$$

$$g_R(-1) = 0, \qquad g_R(1) = 1, \tag{9}$$

and, based on symmetry considerations

$$g_L(\xi) = g_R(-\xi). \tag{10}$$

The corrected gradient $\hat{q}^\delta$ takes the form

$$\hat{q}^\delta = \frac{\partial \hat{u}^\delta}{\partial \xi} + (\hat{u}_L^{\delta I} - \hat{u}_L^\delta)\frac{dg_L}{d\xi} + (\hat{u}_R^{\delta I} - \hat{u}_R^\delta)\frac{dg_R}{d\xi}. \tag{11}$$

where $\hat{u}_L^{\delta I}$ and $\hat{u}_R^{\delta I}$ are the transformed common solutions at the left and right interfaces and $\hat{u}_L^\delta = \hat{u}^\delta(-1)$ and $\hat{u}_R^\delta = \hat{(u)}^\delta(1)$ are the values of the approximate solution at the left and right interfaces, evaluated from equation (7). The exact form of g_L and g_R will be discussed at the end of the subsection.

The fourth stage of the FR approach involves constructing a degree p polynomial $\hat{f}^{\delta D} = \hat{f}^{\delta D}(\xi)$, defined as the approximate transformed discontinuous flux within $\mathbf{\Omega}_S$. A collocation projection at the $p+1$ solution points is employed to obtain $\hat{f}^{\delta D}$, which can hence be expressed as

$$\hat{f}^{\delta D}(\xi) = \sum_{i=1}^{N_s^{1D}} \hat{f}_i^{\delta D}\, l_i(\xi) \tag{12}$$

where the coefficients $\hat{f}_i^{\delta D} = \hat{f}_i^{\delta D}$ are simply the values of the transformed flux at each solution point ξ_i evaluated directly from the approximate solution $\hat{u}^\delta$, and the corrected gradient $\hat{q}^\delta$. The flux $\hat{f}^{\delta D}$ is termed discontinuous since it is calculated from the approximate solution, which is in general piecewise discontinuous between elements.

The fifth stage of the FR approach involves calculating numerical interface fluxes at either end of the standard element $\mathbf{\Omega}_S$ (at $\xi = \pm 1$). In order to calculate these fluxes, one must first obtain values for the approximate solution and the corrected gradient at either end of the standard element via equations (7) and (11), respectively. Once these values have been obtained they can be used in conjunction with analogous information from adjoining elements to calculate numerical interface fluxes. The common numerical interface fluxes associated with the left and right ends of $\mathbf{\Omega}_n$ will be denoted $f_L^{\delta I}$ and $f_R^{\delta I}$, respectively. Their transformed counterparts for use in $\mathbf{\Omega}_S$ will be denoted by $\hat{f}_L^{\delta I}$ and $\hat{f}_R^{\delta I}$ respectively. The exact methodology for calculating such numerical interface fluxes will depend on the nature of the equations being solved. When solving the Navier-Stokes equations, the common numerical interface flux is computed as the sum of an inviscid and viscous part. The inviscid common numerical flux is computed using a Roe type approximate Riemann solver,[21] or any other two-point flux formula that provides for an upwind bias, using $u^{\delta+}$ and $u^{\delta-}$ (the approximate solution on each side of the interface). Following the BR2 approach,[20] the viscous common numerical flux is computed using the average value of the solution $((u^{\delta+} + u^{\delta-})/2)$ and the average value of the corrected gradient $((q^{\delta+} + q^{\delta-})/2)$.

The penultimate stage of the FR approach involves adding a degree $p+1$ transformed correction flux $\hat{f}^{\delta C} = \hat{f}^{\delta C}(\xi)$ to the approximate transformed discontinuous flux $\hat{f}^{\delta D}$, such that their sum equals the transformed numerical interface flux at $\xi = \pm 1$, yet follows (in some sense) the approximate discontinuous flux within the interior of $\mathbf{\Omega}_S$. In order to define $\hat{f}^{\delta C}$ such that it satisfies the above requirements, consider first defining degree $p+1$ correction functions $h_L = h_L(\xi)$ and $h_R = h_R(\xi)$ which are analogous to the correction functions g_L and g_R used to construct the corrected gradient. The correction functions h_L and h_R also approximate zero within $\mathbf{\Omega}_S$, as well as satisfying

$$h_L(-1) = 1, \qquad h_L(1) = 0, \tag{13}$$

$$h_R(-1) = 0, \qquad h_R(1) = 1, \tag{14}$$

and, based on symmetry considerations

$$h_L(\xi) = h_R(-\xi). \tag{15}$$

A suitable expression for $\hat{f}^{\delta C}$ can now be written in terms of h_L and h_R as

$$\hat{f}^{\delta C} = (\hat{f}_L^{\delta I} - \hat{f}_L^{\delta D})h_L + (\hat{f}_R^{\delta I} - \hat{f}_R^{\delta D})h_R, \tag{16}$$

where $\hat{f}_L^{\delta D} = \hat{f}^{\delta D}(-1)$ and $\hat{f}_R^{\delta D} = \hat{f}^{\delta D}(1)$. Using this expression, a degree $p+1$ approximate total transformed flux $\hat{f}^\delta = \hat{f}^\delta(\xi)$ within $\mathbf{\Omega}_S$ can be constructed from the discontinuous and correction fluxes as follows

$$\hat{f}^\delta = \hat{f}^{\delta D} + \hat{f}^{\delta C} = \hat{f}^{\delta D} + (\hat{f}_L^{\delta I} - \hat{f}_L^{\delta D})h_L + (\hat{f}_R^{\delta I} - \hat{f}_R^{\delta D})h_R. \tag{17}$$

The final stage of the FR approach involves calculating the divergence of $\hat{f}^{\delta}$ at each solution point ξ_i using the expression

$$\frac{\partial \hat{f}^{\delta}}{\partial r}(\xi_i) = \sum_{j=1}^{N_s^{1D}} \hat{f}_j^{\delta D} \frac{\mathrm{d}l_j}{\mathrm{d}\xi}(\xi_i) + (\hat{f}_L^{\delta I} - \hat{f}_L^{\delta D}) \frac{\mathrm{d}h_L}{\mathrm{d}\xi}(\xi_i) + (\hat{f}_R^{\delta I} - \hat{f}_R^{\delta D}) \frac{\mathrm{d}h_R}{\mathrm{d}\xi}(\xi_i). \tag{18}$$

These values can then be used to advance the approximate transformed solution $\hat{u}^{\delta}$ in time via a suitable temporal discretization of the following semi-discrete expression

$$\frac{\mathrm{d}\hat{u}_i^{\delta}}{\mathrm{d}t} = -\frac{\partial \hat{f}^{\delta}}{\partial \xi}(\xi_i). \tag{19}$$

The nature of a particular FR scheme depends solely on three factors, namely the location of the solution collocation points ξ_i, the methodology for calculating the common interface solution $u^{\delta I}$ and common numerical interface flux $f^{\delta I}$, and finally the form of the corrected gradient and flux correction functions g_L and h_L (and thus g_R and h_R). Huynh has shown that a collocation based (under integrated) nodal DG scheme is recovered in 1D for the linear advection equation if the correction functions h_L and h_R are the right and left Radau polynomials respectively.[12] Huynh has also shown that SD type methods can be recovered for the linear advection equation if the corrections h_L and h_R are set to zero at a set of p points within $\mathbf{\Omega}_S$ (located symmetrically about the origin).[12] Huynh also suggested several additional forms of h_L (and thus h_R), leading to the development of new schemes, with various stability and accuracy properties. In 2010, Vincent, Castonguay and Jameson[15] identified a class of correction functions h_L and h_R that lead to FR schemes which are linearly stable for all orders of accuracy. Those schemes are referred to as VCJH schemes and they are the ones implemented in the Navier-Stokes solver developed in this work. If the correction functions g_L and g_R (used to construct the correction gradient $\hat{q}$) are chosen as the VCJH correction functions, linear stability for the advection-diffusion equation can also be shown. The proof will be presented in a subsequent publication.

B. FR Approach for Quadrilateral and Hexahedral Elements

The FR framework can easily be extended to quadrilateral and hexahedral elements. In this section, the procedure will be summarized for the quadrilateral element only, but the extension to the hexahedral element is straightforward. More details can be found in the articles by Huynh.[12,13]

Consider the 2D scalar conservation law

$$\frac{\partial u}{\partial t} + \nabla_{xy} \cdot \mathbf{f} = 0 \tag{20}$$

within an arbitrary domain $\mathbf{\Omega}$, where $\mathbf{f} = (f, g)$ where $f = f(u, \nabla u)$ and $g = g(u, \nabla u)$ are the fluxes in the x and y directions respectively. Consider partitioning the domain $\mathbf{\Omega}$ into N non-overlapping, conforming quadrilateral elements $\mathbf{\Omega}_n$ such that

$$\mathbf{\Omega} = \bigcup_{n=1}^{N} \mathbf{\Omega}_n. \tag{21}$$

To facilitate the implementation, each quadrilateral element in the physical domain (x, y) is mapped to a reference element in the transformed space (ξ, η) as shown in figure (2). The transformation can be written as

$$\begin{pmatrix} x \\ y \end{pmatrix} = \sum_{i=1}^{K} M_i(\xi, \eta) \begin{pmatrix} x_i \\ y_i \end{pmatrix} \tag{22}$$

where K is the number of points used to define the shape of the physical element, (x_i, y_i) are the cartesian coordinates of those points and $M_i(\xi, \eta)$ are the shape functions.

After transformation, the evolution of u_n^{δ} within any individual $\mathbf{\Omega}_n$ (and thus the evolution of u^{δ} within $\mathbf{\Omega}$) can be determined by solving

$$\frac{\partial \hat{u}^{\delta}}{\partial t} + \nabla_{\xi\eta} \cdot \hat{\mathbf{f}}^{\delta} = 0 \tag{23}$$

where

$$\hat{u}^\delta = \hat{u}^\delta(\xi,\eta,t) = J_n\, u_n^\delta(\Theta_n(\xi,\eta),t), \tag{24}$$

$$\hat{\mathbf{f}}^\delta = \hat{\mathbf{f}}^\delta(\xi,\eta,t) = (\hat{f}^\delta,\hat{g}^\delta) \tag{25}$$

$$= \left(\frac{\partial y}{\partial \eta} f_n^\delta - \frac{\partial x}{\partial \eta} g_n^\delta, -\frac{\partial y}{\partial \xi} f_n^\delta + \frac{\partial x}{\partial \xi} g_n^\delta\right) \tag{26}$$

and the metric terms J_n, $\frac{\partial x}{\partial \xi}$, $\frac{\partial x}{\partial \eta}$, $\frac{\partial y}{\partial \xi}$ and $\frac{\partial y}{\partial \eta}$ (which depend on the shape of element n) can be evaluated from equation (22).

Inside the reference quadrilateral element, a set of $N_s^{quad} = (p+1)^2$ solution points are defined and are represented by red dots in figure (3). For hexahedral elements, the number of solution points is $N_s^{hex} = (p+1)^3$. They are generated by taking the tensor product of a set of 1D solution points. Furthermore, a set of flux points are defined on the boundary of the reference element. For the quadrilateral element, $(p+1)$ flux points are defined on each edge (represented by blue squares in figure (3)) and for hexahedral elements, $(p+1)^2$ flux points are used on each face. The total number of flux points is thus $N_f^{quad} = 4(p+1)$ for quadrilateral elements and $N_f^{hex} = 6(p+1)^2$ for hexahedral elements.

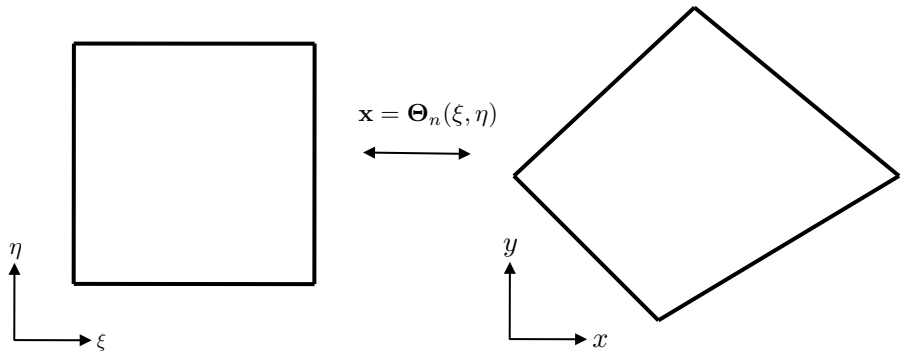

Figure 2: Mapping between the physical space (x, y) and the computational space (ξ, η)

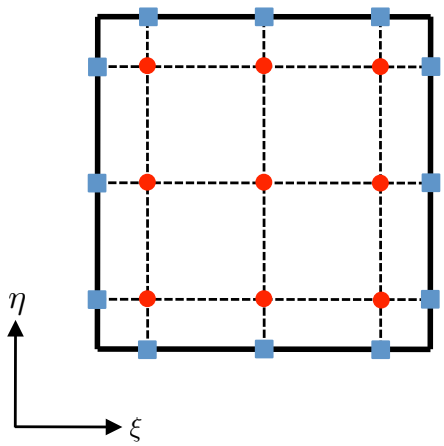

Figure 3: Quadrilateral reference element ($\mathbf{\Omega}_S$) for p=2. Solution points are represented by red circles and flux points by blue squares

Within the reference element, the approximate solution $\hat{u}^\delta$ is written as

$$\hat{u}^\delta = \sum_{i=1}^{N_s^{quad}} \hat{u}_{i,j}^\delta l_i(\xi) l_j(\eta) \tag{27}$$

where $l_i(\xi)$ and $l_j(\eta)$ are 1D Lagrange polynomials associated with the 1D solution points at ξ_i and η_j respectively and $u^{\delta}_{i,j}$ is the value of $\hat{u}^{\delta}$ at the solution point located at (ξ_i, η_j).

The corrected gradient $\hat{\mathbf{q}}^{\delta} = (\hat{q}^{\delta}_{\xi}, \hat{q}^{\delta}_{\eta})$ is obtained using the 1D correction functions g_L and g_R. At each solution point, the ξ and η components of the corrected gradient $\hat{\mathbf{q}}^{\delta} = (\hat{q}^{\delta}_{\xi}, \hat{q}^{\delta}_{\eta})$ are corrected independently as

$$\hat{q}^{\delta}_{\xi}(\xi_i, \eta_j) = \frac{\partial \hat{u}^{\delta}}{\partial \xi}(\xi_i, \eta_j) + (\hat{u}^{\delta I}_L - \hat{u}^{\delta}_L)\frac{\partial g_L}{\partial \xi}(\xi_i) + (\hat{u}^{\delta I}_R - \hat{u}^{\delta}_R)\frac{\partial g_R}{\partial \xi}(\xi_i) \tag{28}$$

$$\hat{q}^{\delta}_{\eta}(\xi_i, \eta_j) = \frac{\partial \hat{u}^{\delta}}{\partial s}(\xi_i, \eta_j) + (\hat{u}^{\delta I}_B - \hat{u}^{\delta}_B)\frac{\partial g_B}{\partial s}(\eta_j) + (\hat{u}^{\delta I}_T - \hat{u}^{\delta}_T)\frac{\partial g_T}{\partial \eta}(\eta_j) \tag{29}$$

where $\hat{u}^{\delta I}_R$, $\hat{u}^{\delta I}_L$, $\hat{u}^{\delta I}_T$, $\hat{u}^{\delta I}_B$ are the transformed common interface values of the approximate solution at the flux points located along the lines $\xi = \xi_i$ and $\eta = \eta_j$ respectively (see figure (4)). The correction functions $g_B(\eta)$ and $g_T(\eta)$ are simply $g_B(\eta) = g_L(\xi = \eta)$ and $g_T(\eta) = g_R(\xi = \eta)$. As shown in figure (4), the gradient at a single solution point depends on the common interface solution values at 4 flux points.

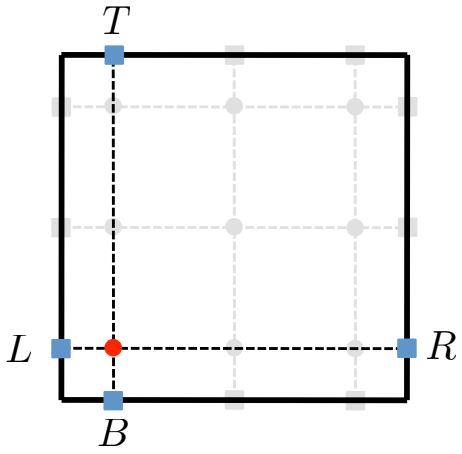

Figure 4: Sketch of the reference quadrilateral element illustrating which flux points contribute to the approximate gradient and approximate divergence of the flux at one solution point

The corrected gradient in the entire element is then constructed as

$$\hat{\mathbf{q}}^{\delta} = \sum_{i=1}^{N_s^{quad}} \hat{\mathbf{q}}^{\delta}_{i,j} l_i(\xi) l_j(\eta). \tag{30}$$

As in the 1D FR approach, the total transformed approximate flux $\hat{\mathbf{f}}^{\delta} = (\hat{f}^{\delta}, \hat{g}^{\delta})$ is written as the sum of a discontinuous component $\hat{\mathbf{f}}^{\delta D}$ and a correction component $\hat{\mathbf{f}}^{\delta C}$,

$$\hat{\mathbf{f}}^{\delta} = \hat{\mathbf{f}}^{\delta D} + \hat{\mathbf{f}}^{\delta C}. \tag{31}$$

The transformed discontinuous flux $\hat{\mathbf{f}}^{\delta D}$ is computed as

$$\hat{\mathbf{f}}^{\delta D}(\xi, \eta) = \sum_{i=1}^{N_s^{quad}} \hat{\mathbf{f}}^{\delta D}_{i,j}\, l_i(\xi)\, l_j(\eta) \tag{32}$$

where the coefficients $\hat{\mathbf{f}}^{\delta D}_{i,j}$ are simply the values of the transformed flux at the solution point (ξ_i, η_j) evaluated directly from the approximate solution $\hat{u}^{\delta}$, and the corrected gradient $\hat{\mathbf{q}}^{\delta}$. The divergence of the discontinuous flux is thus

$$\nabla_{\xi\eta} \cdot \hat{\mathbf{f}}^{\delta D}(\xi, \eta) = \frac{\partial \hat{f}^{\delta D}}{\partial \xi} + \frac{\partial \hat{g}^{\delta D}}{\partial \eta} \tag{33}$$

$$= \sum_{i=1}^{N_s^{quad}} \hat{f}^{\delta D}_{i,j} \frac{\partial l_i(\xi)}{\partial \xi} l_j(\eta) + \sum_{i=1}^{N_s^{quad}} \hat{f}^{\delta D}_{i,j}\, l_i(\xi) \frac{\partial l_j(\eta)}{\partial \eta}. \tag{34}$$

The divergence of the transformed correction flux $\nabla_{\xi\eta} \cdot \hat{\mathbf{f}}^{\delta C} = \frac{\partial \hat{f}^{\delta C}}{\partial \xi} + \frac{\partial \hat{g}^{\delta C}}{\partial \eta}$ at solution point (ξ_i, η_j) is computed following the 1D methodology as

$$\frac{\partial \hat{f}^{\delta C}}{\partial \xi}(\xi_i, \eta_j) = (\hat{f}_L^{\delta I} - \hat{f}_L^{\delta D})\frac{\partial h_L}{\partial \xi}(\xi_i) + (\hat{f}_R^{\delta I} - \hat{f}_R^{\delta D})\frac{\partial h_R}{\partial \xi}(\xi_i) \tag{35}$$

$$\frac{\partial \hat{g}^{\delta C}}{\partial \eta}(\xi_i, \eta_j) = (\hat{g}_B^{\delta I} - \hat{g}_B^{\delta D})\frac{\partial h_B}{\partial \eta}(\eta_j) + (\hat{g}_T^{\delta I} - \hat{g}_T^{\delta D})\frac{\partial h_T}{\partial \eta}(\eta_j) \tag{36}$$

where $h_B(\eta) = h_L(\xi = \eta)$ and $h_T(\eta) = h_R(\xi = \eta)$. In equations (35) and (36), $\hat{f}_L^{\delta I}$, $\hat{f}_R^{\delta I}$, $\hat{g}_B^{\delta I}$ and $\hat{g}_T^{\delta I}$ are the transformed interface numerical fluxes computed at the flux points located along lines $\xi = \xi_i$ and $\eta = \eta_j$. The discontinuous transformed fluxes $\hat{f}_L^{\delta D}$, $\hat{f}_R^{\delta D}$, $\hat{g}_B^{\delta D}$ and $\hat{g}_T^{\delta D}$ are computed from equation (32).

Finally, the solution at the solution points can be updated from

$$\frac{\mathrm{d}\hat{u}_{i,j}^{\delta}}{\mathrm{d}t} = -\frac{\partial \hat{f}^{\delta}}{\partial r}(\xi_i, \eta_j) + \frac{\partial \hat{g}^{\delta}}{\partial s}(\xi_i, \eta_j) \tag{37}$$

where the RHS can be evaluated from equations (31), (32), (35) and (36).

C. FR Approach for Simplex Elements

The FR procedure can be extended to deal with simplex elements (triangles in 2D and tetrahedrals in 3D). In this section, the approach discussed in the article by Castonguay et al.[16] is summarized. In addition, the methodology used to handle diffusion terms is presented. For simplicity, the procedure will be presented for the triangle case, but it can be extended to tetrahedral elements. Consider the 2D scalar conservation law given by equation (20). As before, the domain $\mathbf{\Omega}$ is partitioned into N non-overlapping, conforming triangular elements $\mathbf{\Omega}_n$ such that

$$\mathbf{\Omega} = \bigcup_{n=1}^{N} \mathbf{\Omega}_n. \tag{38}$$

Each element $\mathbf{\Omega}_n$ in physical space is mapped to a reference triangle $\mathbf{\Omega}_S$ using a mapping $\mathbf{\Theta}_n$, as shown in figure (5).

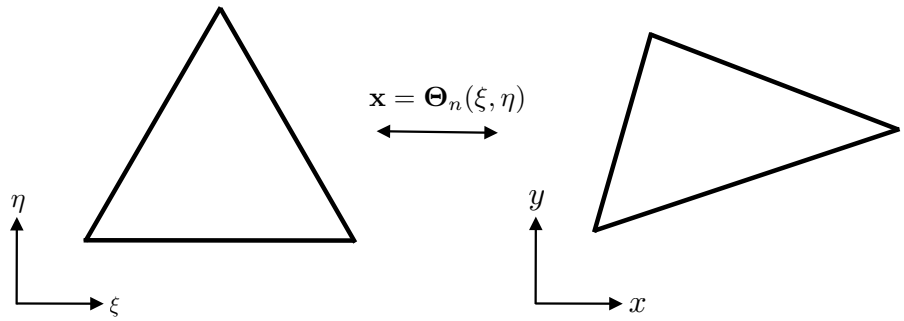

Figure 5: Mapping between the physical space (x, y) and the computational space (ξ, η)

After transformation, the evolution of u_n^{δ} within any individual $\mathbf{\Omega}_n$ (and thus the evolution of u^{δ} within $\mathbf{\Omega}$) can be determined by solving

$$\frac{\partial \hat{u}^{\delta}}{\partial t} + \nabla_{\xi\eta} \cdot \hat{\mathbf{f}}^{\delta} = 0. \tag{39}$$

The approximate solution $\hat{u}^{\delta}$ within the reference triangular element $\mathbf{\Omega}_S$ is represented by a multi-dimensional polynomial of degree p, defined by its values at a set of $N_s^{tri} = \frac{1}{2}(p+1)(p+2)$ solution points (represented by red circles in figure (6)). For tetrahedral elements, the number of solution points is $N_s^{tet} = \frac{1}{6}(p+1)(p+2)(p+3)$.

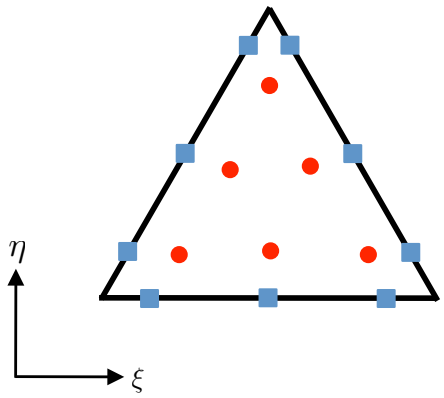

Figure 6: Solution points (red circles) and flux points (blue squares) in the reference element for p= 2

The approximate solution in the reference element takes the form

$$\hat{u}^{\delta}(\boldsymbol{\xi}, t) = \sum_{i=1}^{N_s^{tri}} \hat{u}_i^{\delta} L_i(\boldsymbol{\xi}) \tag{40}$$

where $\boldsymbol{\xi} = (\xi, \eta)$, $\hat{u}_i^{\delta} = J_n \cdot u^{\delta}(\Theta_n^{-1}(\boldsymbol{\xi}_i), t)$ is the value of $\hat{u}^{\delta}$ at the solution point i and $L_i(\boldsymbol{\xi})$ is the multi-dimensional Lagrange polynomial associated with the solution point i in the reference equilateral triangle $\boldsymbol{\Omega}_S$.

The corrected gradient $\hat{\mathbf{q}}_i^{\delta}$ at the solution point located at $\boldsymbol{\xi}_i = (\xi_i, \eta_i)$ is computed from

$$\hat{\mathbf{q}}_i^{\delta} = \left(\nabla_{\xi\eta} \hat{u}^{\delta}\right)\big|_{\boldsymbol{\xi}_i} + \sum_{f=1}^{3} \sum_{j=1}^{p+1} \left(\hat{u}_{f,j}^{\delta I} - \hat{u}_{f,j}^{\delta}\right) \psi_{f,j}|_{\boldsymbol{\xi}_i} \hat{\mathbf{n}}_{f,j} \tag{41}$$

In the previous equation, expressions subscripted by the indices f, j correspond to a quantity at the flux point j of face f, where $1 \leq f \leq 3$ and $1 \leq j \leq (p+1)$. The convention used to number the faces and flux points is illustrated in figure (7).

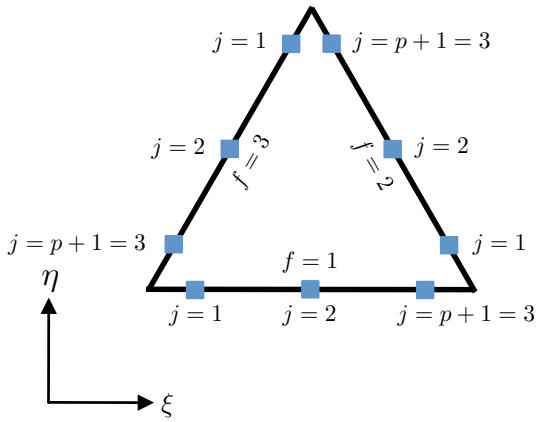

Figure 7: Numbering convention for the faces and flux points on the reference triangular element. Example shown corresponds to $p = 2$.

Each edge of the reference triangle element has $(p + 1)$ flux points while each face of the reference tetrahedral element has $\frac{1}{2}(p+1)(p+2)$ flux points. Thus, the triangle element has a total of $N_f^{tri} = 3(p+1)$ flux points while the tetrahedral element has $N_f^{tet} = 2(p + 1)(p + 2)$ flux points. In equation (41), $\hat{u}_{f,j}^{\delta I}$ is the common interface values of the approximate solution at the flux points f, j. Finally, $\psi_{f,j} = \psi_{f,j}(\xi, \eta)$

is a correction field associated with flux point f, j. The nature of $\psi_{f,j}$ will be discussed at the end of the subsection. The corrected gradient $\hat{\mathbf{q}}^{\delta}$ inside the reference element is constructed as

$$\hat{\mathbf{q}}^{\delta}(\boldsymbol{\xi}) = \sum_{i=1}^{N_s^{tri}} \hat{\mathbf{q}}_i^{\delta} L_i(\boldsymbol{\xi}). \tag{42}$$

As in the 1D FR approach, the total transformed approximate flux $\hat{\mathbf{f}}^{\delta} = (\hat{f}^{\delta}, \hat{g}^{\delta})$ is written as the sum of a discontinuous component $\hat{\mathbf{f}}^{\delta D}$ and a correction component $\hat{\mathbf{f}}^{\delta C}$,

$$\hat{\mathbf{f}}^{\delta} = \hat{\mathbf{f}}^{\delta D} + \hat{\mathbf{f}}^{\delta C}. \tag{43}$$

The transformed discontinuous flux $\hat{\mathbf{f}}^{\delta D} = (\hat{f}^{\delta D}, \hat{g}^{\delta D})$ is computed by constructing a degree p polynomial for each of its components as follows

$$\hat{\mathbf{f}}^{\delta D} = \sum_{i=1}^{N_s^{tri}} \hat{\mathbf{f}}_i^{\delta D} L_i \tag{44}$$

where $\hat{\mathbf{f}}_i^{\delta D}$ is the value of the transformed flux at the solution point i evaluated directly from the approximate solution $\hat{u}_i$ and the approximate corrected gradient $\hat{\mathbf{q}}_i$ (i.e. $\hat{\mathbf{f}}_i^{\delta D} = \hat{\mathbf{f}}(\hat{u}_i, \hat{\mathbf{q}}_i)$). The divergence of the transformed discontinuous flux is therefore

$$\nabla_{\xi\eta} \cdot \hat{\mathbf{f}}^{\delta D} = \sum_{i=1}^{N_s^{tri}} \hat{f}_i^{\delta D} \frac{\partial L_i}{\partial \xi} + \sum_{i=1}^{N_s^{tri}} \hat{g}_i^{\delta D} \frac{\partial L_i}{\partial \eta} \tag{45}$$

The transformed correction flux $\hat{\mathbf{f}}^{\delta C}$ is constructed as follows

$$\hat{\mathbf{f}}^{\delta C}(\boldsymbol{\xi}) = \sum_{f=1}^{3} \sum_{j=1}^{p+1} \left[(\hat{\mathbf{f}} \cdot \hat{\mathbf{n}})_{f,j}^{\delta I} - (\hat{\mathbf{f}}^{\delta D} \cdot \hat{\mathbf{n}})_{f,j} \right] \mathbf{h}_{f,j}(\boldsymbol{\xi}) \tag{46}$$

$$\hat{\mathbf{f}}^{\delta C}(\boldsymbol{\xi}) = \sum_{f=1}^{3} \sum_{j=1}^{p+1} \Delta_{f,j} \mathbf{h}_{f,j}(\boldsymbol{\xi}) \tag{47}$$

In equation (46), $(\hat{\mathbf{f}}^{\delta D} \cdot \hat{\mathbf{n}})_{f,j}$ is the normal component of the transformed discontinuous flux $\hat{\mathbf{f}}^{\delta D}$ at the flux point f, j and $(\hat{\mathbf{f}} \cdot \hat{\mathbf{n}})_{f,j}^{\delta I}$ is a transformed normal numerical flux computed at flux point f, j. Finally, $\mathbf{h}_{f,j}(\boldsymbol{\xi})$ is a vector correction function associated with flux point f, j. Each vector correction function $\mathbf{h}_{f,j}(\boldsymbol{\xi})$ is restricted to lie in the Raviart-Thomas space[22] of order p. Because of this property, the divergence of each correction function $(\nabla_{\xi\eta} \cdot \mathbf{h}_{f,j})$ is a polynomial of degree p and the normal trace $\mathbf{h}_{f,j} \cdot \hat{\mathbf{n}}$ is also a polynomial of degree p along each edge. The latter property is required to prove that the resulting scheme is conservative. The correction functions $\mathbf{h}_{f,j}$ satisfy

$$\mathbf{h}_{f,j}(\boldsymbol{\xi}_{f_2,j_2}) \cdot \mathbf{n}_{f_2,j_2} = \begin{cases} 1 & \text{if } f = f_2 \text{ and } j = j_2 \\ 0 & \text{if } f \neq f_2 \text{ or } j \neq j_2 \end{cases} \tag{48}$$

An example of a vector correction function $\mathbf{h}_{f,j}$ is shown in figure (8) for the case $p = 2$.

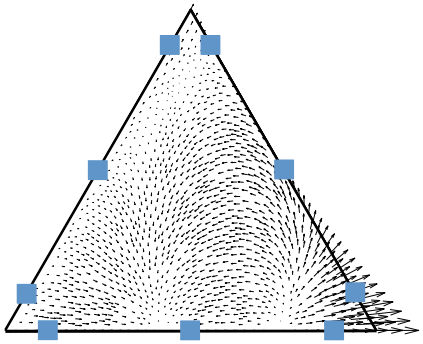

Figure 8: Example of a vector correction function $\mathbf{h}_{f,j}$ associated with flux point $f = 2, j = 1$ for $p = 2$

The correction field $\phi_{f,j}(\boldsymbol{\xi})$ is defined as the divergence of the correction function $\mathbf{h}_{f,j}(\boldsymbol{\xi})$, i.e.

$$\phi_{f,j}(\boldsymbol{\xi}) = \nabla_{\xi\eta} \cdot \mathbf{h}_{f,j}(\boldsymbol{\xi}). \tag{49}$$

Finally, combining equations (39), (43), (45) and (47), the approximate solution values at the solution points can be updated from

$$\begin{aligned}
\frac{d\hat{u}_i^\delta}{dt} &= -\left(\nabla_{\xi\eta} \cdot \hat{\mathbf{f}}^\delta\right)\Big|_{\boldsymbol{\xi}_i} && (50)\\
&= -\left(\nabla_{\xi\eta} \cdot \hat{\mathbf{f}}^{\delta D}\right)\Big|_{\boldsymbol{\xi}_i} - \left(\nabla_{\xi\eta} \cdot \hat{\mathbf{f}}^{\delta C}\right)\Big|_{\boldsymbol{\xi}_i} && (51)\\
&= -\sum_{k=1}^{N_s^{tri}} \hat{f}_k^{\delta D} \frac{\partial L_k}{\partial \xi}\Big|_{\boldsymbol{\xi}_i} - \sum_{k=1}^{N_s^{tri}} \hat{g}_k^{\delta D} \frac{\partial L_k}{\partial \eta}\Big|_{\boldsymbol{\xi}_i} - \sum_{f=1}^{3}\sum_{j=1}^{(p+1)} \Delta_{f,j}\phi_{f,j}(\boldsymbol{\xi}_i). && (52)
\end{aligned}$$

The nature of a particular FR scheme on triangular elements depends on four factors, namely the location of the solution points $\boldsymbol{\xi}_i$, the location of the flux points $\boldsymbol{\xi}_{f,j}$, the methodology for calculating the common interface solutions $u^{\delta I}$ and numerical interface fluxes $(\mathbf{f}\cdot\mathbf{n})_{f,j}^{\delta I}$ and finally the form of the correction fields $\psi_{f,j}$ and $\phi_{f,j}$. In reference 16, Castonguay et al. identify a range of correction fields $\phi_{f,j}$ that are guaranteed to yield a linearly stable scheme for all orders of accuracy and can also provide an increased time step limit compared to the DG scheme. These schemes are referred to as VCJH schemes on triangles. If the correction fields $\psi_{f,j}$ (used to construct the correction gradient $\hat{\mathbf{q}}$) are chosen as the VCJH correction fields, linear stability for the advection-diffusion equation can also be shown. The proof will be presented in a subsequent publication.

D. FR Approach for Prismatic Elements

To extend the approach to prismatic elements, a tensor-product combination of the FR approaches described in subsections A and C is used. The approximate solution $\hat{u}^\delta$ within the reference prismatic element $\boldsymbol{\Omega}_S$ is represented by a multi-dimensional polynomial defined by its values at a set of $N_s^{pri} = N_s^{tri}(p+1) = \frac{1}{2}(p+1)(p+2)(p+1)$ solution points (represented by red circles in figure (9)). The solution points are numbered using two indices (i,j), the index i indicates the position of the point on the triangle in the ξ,η plane and the index j indicates the position along the ζ-direction. Thus, the solution point with index (i,j) is located at (ξ_i,η_i,ζ_j).

The approximate solution in the reference element takes the form

$$\hat{u}^\delta(\xi,\eta,\zeta) = \sum_{i=1}^{N_p^{tri}}\sum_{j=1}^{p+1} \hat{u}_{i,j}^\delta L_i(\xi,\eta) l_j(\zeta) \tag{53}$$

where $\hat{u}_{i,j}^\delta$ is the value of $\hat{u}^\delta$ at the solution point (i,j), $L_i(\xi,\eta)$ is the 2D dimensional Lagrange polynomial associated with the solution point i in the reference triangle, and $l_j(\zeta)$ is the one-dimensional Lagrange polynomial associated with the solution point located at ζ_j.

The corrected gradient $\hat{\mathbf{q}}^\delta = (\hat{q}_\xi^\delta, \hat{q}_\eta^\delta, \hat{q}_\zeta^\delta)$ is computed in a decoupled manner. The ξ and η components of the correction gradient at the solution points are computed using the methodology used on triangles, i.e.

$$\hat{q}_\xi^\delta\big|_{\boldsymbol{\xi}_{i,j}} = \frac{\partial \hat{u}^\delta}{\partial \xi}\Big|_{\boldsymbol{\xi}_{i,j}} + \sum_{f=1}^{3}\sum_{k=1}^{p+1} \left(\hat{u}_{f,k,j}^{\delta I} - \hat{u}_{f,k,j}^{\delta D}\right) \psi_{f,k}|_{\boldsymbol{\xi}_{i,j}} (\hat{n}_\xi)_{f,k,j} \tag{54}$$

$$\hat{q}_\eta^\delta\big|_{\boldsymbol{\xi}_{i,j}} = \frac{\partial \hat{u}^\delta}{\partial \eta}\Big|_{\boldsymbol{\xi}_{i,j}} + \sum_{f=1}^{3}\sum_{k=1}^{p+1} \left(\hat{u}_{f,k,j}^{\delta I} - \hat{u}_{f,k,j}^{\delta D}\right) \psi_{f,k}|_{\boldsymbol{\xi}_{i,j}} (\hat{n}_\eta)_{f,k,j} \tag{55}$$

where the subscripts f,k refers to the position of the flux point in the reference triangle element, and the subscript j refers to the position of the flux point in the ζ direction. The ζ component of the corrected gradient at the solution points is computed using the 1D correction functions h_T and h_B, i.e.

$$\hat{q}_\zeta^\delta\big|_{\boldsymbol{\xi}_{i,j}} = \frac{\partial \hat{u}^\delta}{\partial \zeta}\Big|_{\boldsymbol{\xi}_{i,j}} + (\hat{u}_{T,i}^{\delta I} - \hat{u}_{T,i}^\delta)\frac{dh_T}{d\zeta}(\zeta_j) + (\hat{u}_{B,i}^{\delta I} - \hat{u}_{B,i}^\delta)\frac{dh_B}{d\zeta}(\zeta_j). \tag{56}$$

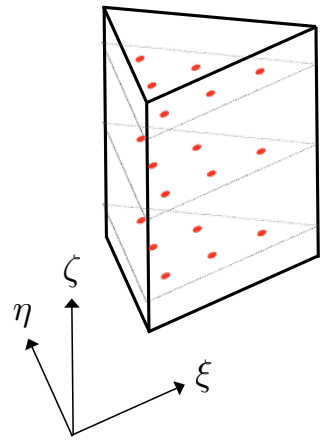

Figure 9: Solution points, represented by red circles in the reference prismatic element for p= 2

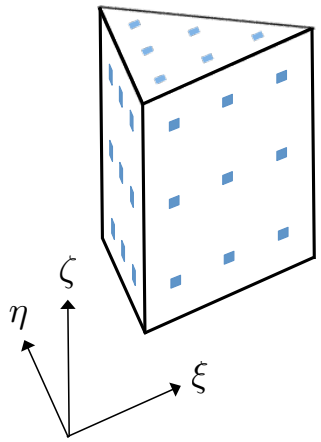

Figure 10: Flux points represented by blue squares in the reference prismatic element for p= 2

where the subscript i in $\hat{u}^{\delta}_{T,i}$, $\hat{u}^{\delta}_{B,i}$, $\hat{u}^{\delta I}_{T,i}$ and $\hat{u}^{\delta I}_{B,i}$ refers to the position of the flux point on the triangular faces at the top (T) and bottom (B) of the prismatic element. The corrected gradient in the rest of the element takes the form

$$\hat{\mathbf{q}}^{\delta}(\xi,\eta,\zeta) = \sum_{i=1}^{N_p^{tri}} \sum_{j=1}^{p+1} \hat{\mathbf{q}}^{\delta}_{i,j} L_i(\xi,\eta) l_j(\zeta) \tag{57}$$

The discontinuous approximate flux is

$$\hat{\mathbf{f}}^{\delta D}(\xi,\eta,\zeta) = \sum_{i=1}^{N_p^{tri}} \sum_{j=1}^{p+1} \hat{\mathbf{f}}^{\delta D}_{i,j} L_i(\xi,\eta) l_j(\zeta). \tag{58}$$

Finally, the divergence of the correction flux at the solution points takes the form

$$\nabla_{\xi\eta\zeta} \cdot \hat{\mathbf{f}}^{\delta C}\Big|_{\boldsymbol{\xi}_{i,j}} = \sum_{f=1}^{3} \sum_{k=1}^{p+1} \left[(\hat{\mathbf{f}} \cdot \hat{\mathbf{n}})^{\delta I}_{f,k,j} - (\hat{\mathbf{f}}^{\delta D} \cdot \hat{\mathbf{n}})_{f,k,j} \right] \phi_{f,k}(\xi_i, \eta_i) \tag{59}$$

$$+ \left[(\hat{\mathbf{f}} \cdot \hat{\mathbf{n}})^{\delta I}_{T,i} - (\hat{\mathbf{f}} \cdot \hat{\mathbf{n}})^{\delta D}_{T,i} \right] \frac{dh_T}{d\zeta}(\zeta_j) \tag{60}$$

$$+ \left[(\hat{\mathbf{f}} \cdot \hat{\mathbf{n}})^{\delta I}_{B,i} - (\hat{\mathbf{f}} \cdot \hat{\mathbf{n}})^{\delta D}_{B,i} \right] \frac{dh_B}{d\zeta}(\zeta_j) \tag{61}$$

where $(\hat{\mathbf{f}} \cdot \hat{\mathbf{n}})^{\delta I}_{f,k,j}$ are the transformed common numerical interface fluxes on the side faces of the reference prismatic element (faces with $\hat{n}_\zeta = 0$) and $(\hat{\mathbf{f}} \cdot \hat{\mathbf{n}})^{\delta I}_{T,i}$ and $(\hat{\mathbf{f}} \cdot \hat{\mathbf{n}})^{\delta I}_{B,i}$ are the transformed common numerical interfaces fluxes on the top and bottom faces.

E. Extension to Navier-Stokes Equations

The FR procedure described in the previous sections can be used to solve the Navier-Stokes equations. The GPU algorithm presented in this work solves the unsteady, three-dimensional, compressible, unfiltered Navier-Stokes equations which can be expressed as

$$\frac{\partial U}{\partial t} + \nabla \cdot \mathbf{F} = 0 \tag{62}$$

where $\mathbf{F} = (F, G, H) = (F_I, G_I, H_I) - (F_V, G_V, H_V)$. The state vector U, inviscid flux vectors F_I,G_I and H_I, along with the viscous flux vectors F_V,G_V and H_V are

$$U = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho e \end{pmatrix} F_I = \begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho uw \\ \rho eu + p \end{pmatrix}, G_I = \begin{pmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ \rho vw \\ \rho ev + p \end{pmatrix}, H_I = \begin{pmatrix} \rho w \\ \rho uw \\ \rho uw \\ \rho w^2 + p \\ \rho ew + p \end{pmatrix} \tag{63}$$

$$F_V = \begin{pmatrix} 0 \\ \sigma_{xx} \\ \sigma_{xy} \\ \sigma_{xz} \\ u_i\sigma_{ix} - q_x \end{pmatrix}, G_V = \begin{pmatrix} 0 \\ \sigma_{yx} \\ \sigma_{yy} \\ \sigma_{yz} \\ u_i\sigma_{iy} - q_y \end{pmatrix}, H_V = \begin{pmatrix} 0 \\ \sigma_{zx} \\ \sigma_{zy} \\ \sigma_{zz} \\ u_i\sigma_{iz} - q_z \end{pmatrix}. \tag{64}$$

In these definitions, ρ is the density, u, v and w are the velocity components in the x, y and z directions respectively and e is the total energy per unit mass. The pressure is determined from the equation of state,

$$p = (\gamma - 1)\rho\left(e - \frac{1}{2}(u^2 + v^2 + w^2)\right). \tag{65}$$

For a Newtonian fluid, the viscous stresses are

$$\sigma_{ij} = \mu\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) - \frac{2}{3}\mu\delta_{ij}\frac{\partial u_k}{\partial x_k} \tag{66}$$

and the heat fluxes are

$$q_i = -k\frac{\partial T}{\partial x_i}. \tag{67}$$

The coefficient of thermal conductivity and the temperature are computed as

$$k = \frac{C_p\mu}{Pr}, \quad T = \frac{p}{R\rho} \tag{68}$$

where Pr is the Prandtl number, C_p is the specific heat at constant pressure and R is the gas constant. For the cases considered in this paper, $\gamma = 1.4$ and $Pr = 0.72$. The dynamic viscosity μ can be computed from Sutherland's law as

$$\mu = \mu_{ref}\left(\frac{T}{T_{ref}}\right)^{\frac{3}{2}}\frac{T_{ref} + S}{T + S} \tag{69}$$

Note that $\mathbf{F}$ is a function of U and ∇U and therefore the methodology presented in the previous sections can be applied, the main difference being that the Navier-Stokes equations are a coupled system of 5 equations instead of just one scalar equation.

III. Overview of the steps in the FR approach

In this section, the steps required to update the solution at one solution point are summarized and are formulated in a way that applies to all element types. The objective of this section is to provide insights into how the steps in the FR approach are implemented on the GPU. It should also be noted that if the solution points and flux points are overlapped, many of the steps can be neglected. However, theoretical analysis[18] and numerical experiments[19] suggests that locating the solution points at good volume quadrature points and the flux points at good face quadrature points is essential to prevent instabilities due to aliasing errors.

Step 1: Compute the approximate solution at the flux points

First, for each element, the approximate transformed solution must be computed at the flux points located at the cell interfaces. Consider defining a matrix $[\hat{U}_s]$ of dimension $N_s \times 5$ (where N_s is the number of solution

points per cell and 5 is the number of equations in the 3D compressible Navier-Stokes equations) which stores the approximate solution at the solution points of a cell. Furthermore, consider defining a matrix $[\hat{U}_f]$ of dimension $N_f \times 5$ (where N_f is the number of flux points per cell) that stores the approximate solution at the flux points of a cell. For the Navier-Stokes equations, the matrices $[\hat{U}_s]$ and $[\hat{U}_f]$ have the form

$$[\hat{U}_s] = \begin{bmatrix} \hat{\rho}_1^\delta & \widehat{\rho u}_1^\delta & \widehat{\rho v}_1^\delta & \widehat{\rho w}_1^\delta & \widehat{\rho e}_1^\delta \\ \hat{\rho}_2^\delta & \widehat{\rho u}_2^\delta & \widehat{\rho v}_2^\delta & \widehat{\rho w}_2^\delta & \widehat{\rho e}_2^\delta \\ . & . & . & . & . \\ . & . & . & . & . \\ \hat{\rho}_{N_s}^\delta & \widehat{\rho u}_{N_s}^\delta & \widehat{\rho v}_{N_s}^\delta & \widehat{\rho w}_{N_s}^\delta & \widehat{\rho e}_{N_s}^\delta \end{bmatrix} \tag{70}$$

and

$$[\hat{U}_f] = \begin{bmatrix} \hat{\rho}_1^\delta & \widehat{\rho u}_1^\delta & \widehat{\rho v}_1^\delta & \widehat{\rho w}_1^\delta & \widehat{\rho e}_1^\delta \\ \hat{\rho}_2^\delta & \widehat{\rho u}_2^\delta & \widehat{\rho v}_2^\delta & \widehat{\rho w}_2^\delta & \widehat{\rho e}_2^\delta \\ . & . & . & . & . \\ . & . & . & . & . \\ \hat{\rho}_{N_f}^\delta & \widehat{\rho u}_{N_f}^\delta & \widehat{\rho v}_{N_f}^\delta & \widehat{\rho w}_{N_f}^\delta & \widehat{\rho e}_{N_f}^\delta \end{bmatrix}. \tag{71}$$

To compute the approximate solution $\hat{u}^\delta$ at the flux points, one must perform

$$[\hat{U}_f^D] = M_1[\hat{U}_s] \tag{72}$$

at the cell level, where the matrix M_1 of dimension $N_f \times N_s$ depends on the type of element used and is the same for all elements of the same type. For tetrahedral elements, M_1 is dense while for hexahedral and prismatic elements M_1 is sparse because of their tensor product formulation. In the following steps, the subscript s will be used to indicate a quantity stored at the solution points, and the subscript f to indicate a quantity stored at the flux points.

Step 2: Compute the discontinuous solution gradient $\hat{\nabla}\hat{u}^\delta$

The corrected gradient $\hat{\mathbf{q}}^\delta$ is the sum of a discontinuous solution gradient $\hat{\nabla}\hat{u}^\delta$ and a correction, which depends on a common solution value at the cell interfaces. Consider defining the matrix $[\hat{Q}_s^D]$ of dimension $(3N_s) \times 5$ which stores the discontinuous gradient $\hat{\nabla}\hat{u}^\delta$ at the solution points. The discontinuous solution gradient is obtained from

$$[\hat{Q}_s^D] = M_2[\hat{U}_s] \tag{73}$$

where M_2 is of dimension $3N_s \times N_s$. Again, the structure of M_2 depends on the type of element used and is the same for all elements of the same type.

Step 3: Compute the common interface values $u^{\delta I}$ *and the inviscid part of the common interface flux* $(\mathbf{f}\cdot\mathbf{n})^{\delta I}$ *for each flux point pair*

In this step, a common value of the solution and the numerical inviscid flux must be computed for each flux point pair at the cell interfaces. In the algorithm, this step involves looping over all the face flux point pairs, loading the data on the left and right sides of the interface, computing the common interface solution $u^{\delta I}$ and common normal interface inviscid flux $(\mathbf{f}_{inv}\cdot\mathbf{n})^{\delta I}$ and storing the results. For each cell, let the matrices $[\hat{U}_f^I]$ and $[(\hat{F}\cdot\hat{n})_f^I]$ contain the transformed common interface solution $\hat{u}^{\delta I}$ and the transformed common normal interface flux $(\hat{\mathbf{f}}\cdot\hat{\mathbf{n}})^{\delta I}$. Both matrices are of dimension $N_f \times 5$.

Step 4: Compute the corrected gradient $\hat{\mathbf{q}}^\delta$ *at the solution points*

After the transformed common approximate solution value has been computed at each flux point pair, the corrected gradient $\hat{\mathbf{q}}^\delta$ is computed at the solution points. Within each element, this is represented by a matrix-matrix multiplication of the form

$$[\hat{Q}_s] = M_3\left([\hat{U}_f^I] - [\hat{U}_f^D]\right) + [\hat{Q}_s^D] \tag{74}$$

where the matrix M_3 is of dimension $3N_s \times N_f$.

Step 5: Compute the transformed discontinuous flux $\hat{\mathbf{f}}^{\delta D}$ at the solution points

The transformed discontinuous flux at the solution points depends on the approximate solution $\hat{u}^{\delta}$ and the corrected gradient $\hat{\mathbf{q}}$ which were computed in the previous steps. This operation is performed independently for each solution point in the mesh. For each cell, the result is stored in a matrix of dimension $3N_s \times 5$ denoted by $[\hat{F}_s^D]$.

Step 6: Compute the discontinuous flux divergence $\nabla \cdot \hat{f}^{\delta D}$

The discontinuous flux divergence is computed as

$$[(div\ \hat{F})_s^D] = M_4[\hat{F}_s^D] \tag{75}$$

where the matrix M_4 is of dimension $N_s \times (3N_s)$.

Step 7: Compute the corrected gradient $\hat{\mathbf{q}}$ at the flux points

In order to evaluate the common viscous interface flux at the flux points, the corrected gradient must first be evaluated at the flux points. This is done using matrix M_1 as follows

$$[\hat{Q}_f] = \begin{bmatrix} M_1 & 0 & 0 \\ 0 & M_1 & 0 \\ 0 & 0 & M_1 \end{bmatrix} [\hat{Q}_s] \tag{76}$$

Step 8: Compute the normal discontinuous flux $(\hat{\mathbf{f}}^{\delta D} \cdot \hat{\mathbf{n}})$ at the flux points

The normal discontinuous flux at the flux points, which will be denoted by the matrix $[(\hat{F} \cdot \hat{n})_f^D]$, is required to form the correction flux $\hat{\mathbf{f}}^{\delta C}$. It is computed from

$$[(\hat{F} \cdot \hat{n})_f^D] = M_5[\hat{F}_s^D] \tag{77}$$

where M_5 is of dimension $N_f \times 3N_s$.

Step 9: Compute the viscous part of the common interface flux $(\hat{\mathbf{f}} \cdot \hat{\mathbf{n}})^{\delta I}$ at the flux points

In this step, a common value of the numerical viscous flux must be computed for each flux point pair at the cell interfaces. As in step 3, this step involves looping over all the face flux point pairs, loading the data on the left and right sides of the interface, computing a common normal interface viscous flux $(\mathbf{f}_{vis} \cdot \mathbf{n})^{\delta I}$ and adding the results to the normal interface inviscid flux computed in step 3 and stored in $[(\hat{F} \cdot \hat{n})_f^I]$.

Step 10: Correct the divergence of the discontinuous flux

In this penultimate step, the divergence of the total approximate flux can be computed by adding the divergence of the correction flux $\nabla \cdot \hat{\mathbf{f}}^{\delta C}$ to the divergence of the discontinuous flux $\nabla \cdot \hat{\mathbf{f}}^{\delta D}$. Using the matrices introduced in the previous steps, this is represented as

$$[(div\ \hat{F})_s] = M_6 \left([(\hat{F} \cdot \hat{n})_f^I] - [(\hat{F} \cdot \hat{n})_f^D] \right) + [(div\ \hat{F})_s^D] \tag{78}$$

where M_6 is an operator matrix of dimension $N_f \times N_s$.

Step 11: Using the corrected divergence, update the solution at solution points.

Finally, a time integration scheme (such as the 5-stage, 4th order accurate Runga-Kutta scheme[23] used in this work) is used to update the solution at the solution points, using the previously computed divergence of the flux at the solution points. This step can be performed independently for each solution point in the mesh.

IV. CUDA Overview

Our GPU implementation uses the CUDA architecture from NVIDIA and was designed to run on the newest Fermi architecture. The cases presented in this work were run on the Tesla C2050 GPU which is a single chip GPU with 448 cores and 3GB of memory. The C2050 supports Error Correction Code (ECC) protection for DRAM and IEEE-compliant double precision math. It has a peak double precision floating point performance of 515 Gflops. The chip in the Tesla C2050 has 14 streaming multi-processors (SMs) and each SM contains 32 cores, for a total of 448 cores. On a GPU, a parallel code is executed by so-called threads and the SM is responsible for creating, managing and executing those threads in groups of 32 parallel threads called warps.

The Tesla C2050 card has 3GB of global off-chip DRAM to which all threads have access to. The global off-chip GPU memory and the host CPU's memory have separate memory spaces and the data can be moved between the two via the PCIE bus. On the Tesla C2050 card, the memory bandwidth between the GPU global memory and the GPU chip is about 126 GB/sec with ECC turned on. However, this peak bandwidth can be attained only for certain memory access patterns. In this work, caching loads are used, hence if threads in a warp read from the same 128-byte memory region in the same cycle, these reads are batched into a single operation via a process known as memory coalescing. On the other hand, if threads in a warp access global memory in a random fashion, up to 31/32 of the available memory bandwidth can be wasted. Therefore, in order not to waste memory bandwidth, threads in the same warp must access sequential memory locations at the same time. In addition to the global off-chip memory, the Tesla C2050 has a read-only on-chip cache called texture memory, and a configurable 64KB shared memory/L1 cache which can be configured as 16 KB or 48 KB of L1 cache with the remaining space taken up by shared memory.

The CUDA architecture may be programmed via the programming language CUDA C, which extends the C/C++ language to allow the management of the GPU's memory and the execution of kernels (which are functions running on the GPU). A kernel is executed by threads, which are organized into a two-level hierarchy. Individual threads are grouped into batches of up to 1024 called thread blocks. Threads within the same thread block are guaranteed to run on the same SM at the same time. However, thread blocks can run in any order, so they must be independent. The set of all thread blocks is called a thread grid and all threads in a grid must execute the same kernel. A grid of threads is launched via a modified function call syntax that specifies the kernel function name, its parameters, the number of threads in each block, and the number of blocks in the grid. It should be noted that the shared memory/L1 cache configuration can be set per kernel, allowing the programmer to find the best configuration for a given kernel. For more details on the CUDA programming model, see the CUDA programming guide.[24]

V. Single GPU Implementation

The CFD code developed in this work can be split in three main phases. First, in the setup phase, the input and mesh files are read, the operating matrices (M_1 to M_6) are created and the initial conditions are set. In the computational phase, the steps presented in section III are performed to advance the simulation in time. Finally, at regular intervals during the computation, the data is written to the disk for post-processing, in a phase termed the output phase.

The total time required by the setup and output phases is very small compared to the computational phase (typically less than 0.1%) and therefore, they were not ported to the GPU. The computational phase (which consists of all the steps presented in section III and which dominates the total running time) was ported entirely to the GPU. This was done in order to avoid data transfers across the PCIE bus at each iteration. The PCIE x16 Gen 2 runs at a peak bandwidth of 8 GB/sec, hence even small data transfers at each iteration would dramatically offset any performance improvement on the GPU. Thus, after the setup phase, all data is copied from the CPU to the GPU and stays on the GPU for the entire run of the algorithm. At regular intervals, the data is copied to the CPU for writing to the disk. By using the asynchronous transfer API, this can be done without slowing down the GPU computation.

The steps presented in section III can be grouped into three main types of operations: element-local, point-local and interface operations.

Steps 1, 2, 4, 6, 7, 8 and 10 are performed in an element-local fashion, with no element-to-element coupling. In these steps, data is loaded and used multiple times, hence they tend to have a large instruction to memory fetches ratio. For example, in step 1, the approximate solution at each flux points depends on

the approximate solution at multiple solution points inside the cell, hence the data stored at the solution points is reused multiple times. Furthermore, these steps are computationally intensive.

Steps 5 and 11 are point-local. In these steps, the output quantity depends on the data stored a single solution point. For example, in step 11, the residual at a single solution point is used to advance the solution at the same solution point. These operations benefit from the massively parallel nature of GPUs as they can be executed in parallel for all solution points in the mesh.

Finally, steps 3 and 9 are operations performed at the interfaces between cells. In these steps, for each interface flux point pair, data from neighbouring cells is loaded, a common interface value is computed and written back at the cell level.

The GPU implementation of these three types of operations is discussed in the next subsections. To limit the scope of the article, not all implementation details are presented.

A. Element-local Operations

As discussed in section III, a large fraction of the steps in the FR method can be represented as a matrix-matrix multiplication performed at the cell level. This typical matrix-matrix multiplication operation will be represented by

$$AB = C \tag{79}$$

where the matrix A is the operator matrix (of size $M_A \times K_A$), the matrix B holds the field data that will be operated on (of size $K_A \times N_B$) and C is the resulting matrix (of size M_A by N_B). For the matrix-matrix multiplication steps presented in section III, the number of columns of matrices B and C (denoted by N_B) is 5 (the number of states in the solution vector U). For tetrahedral elements, the operator matrices (M_1 to M_6) are dense while for hexahedral and prismatic elements, those matrices are sparse because of their tensor product formulation. The size of the operator matrices is of the order of the number of solution points or the number of flux points per cell.

These matrix-matrix operations could have been performed on the GPU using the CUBLAS library.[25] However, it was found that a custom matrix-matrix multiplication algorithm that makes use of texture and shared memory outperforms CUBLAS. This can be attributed to three factors. First, the use of custom kernels allows a reduction in the number of global memory fetches since the kernels can be altered to modify the input matrix B or the output matrix C inside the matrix multiplication kernel without having to launch an additional kernel. Second, the operator matrices (M_1 to M_6) are constant throughout the computation and are the same for all cells of the same type, hence in our GPU implementation, they are bound to the small amount of read-only cached memory called texture memory. It was found that the use of texture memory leads to a significant increase in performance. Profiling of the code using the CUDA compute profiler indicates that the texture binding is very effective as the texture hit rate is always higher than 95%. Third, because of the large amount of data reuse when performing a matrix-matrix multiplication, the fast on-chip shared memory (which has two orders of magnitude lower latency than global memory) was used to store the matrix B, which contains the field data to be acted on. This strategy was adopted by Klockner et al.[17] in their GPU implementation of the DG scheme for Maxwell's equations. Numerical experiments indicate that the use of custom kernels to perform the cell-local operations leads to an increase in performance of up to 40% for those kernels, compared to the CUBLAS (version 3.2) implementation.

For the matrix-matrix multiplication kernels, one thread is assigned to each row of the output matrix C. This approach leads to a slightly higher number of registers than when one thread is assigned per entry of the output matrix. However, it reduces the number of global memory fetches for matrix A, and also allows for instruction level parallelism. For each matrix multiplication kernel, a number of cells is assigned per thread block, a quantity which will be denoted by K_m. Thus for each matrix-matrix multiplication kernel, the number of threads is $K_m \cdot M_A$. Because the size of the matrices differ for the various cell-local steps, we allow the number of cells per block to vary from one kernel to the next. This allows us to choose an optimum value of K_m for each matrix-matrix multiplication kernel in the algorithm. Using the CUDA compute profiler, the optimal number of cells per thread block can be identified for all orders and all element types. One consequence of allowing the number of cells per block to vary is that the data cannot be padded to ensure that global memory fetches are always aligned for all the matrix-matrix multiplication kernels. However, it was found that the added flexibility to vary the number of cells per block for the different kernels outweighs the penalty due to unaligned loads. In fact, thanks to the L1 and L2 caches on the newest

Fermi architecture, sequential non-aligned memory loads only suffer from a slight decrease in performance compared to sequential aligned memory loads. Since the matrix-matrix multiplication kernels tend to be compute bound, the impact is limited. Because of the choice of computational layout (and also to ensure coalesced memory accesses for the point-local operations presented in the next subsection), quantities at the solution points or flux points for all the cells of the same type are stored as shown in figure (11). The extra padding is introduced to ensure fully coalesced loads for the point-local operations (steps 5 and 11).

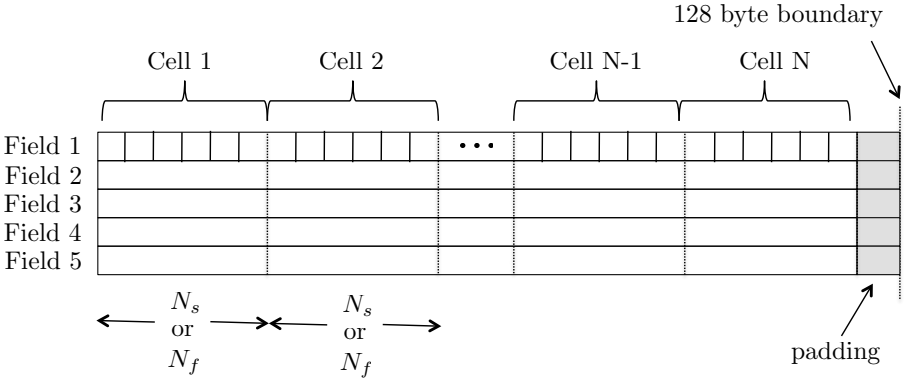

Figure 11: Data layout used to store the field variables

1. Dense matrix-matrix multiplication

For the tetrahedral element, the operator matrices (M_1 to M_6) are dense, hence the element-local operations can be represented by a dense matrix-matrix multiplication. Because one thread is assigned per row of the matrix C, the matrix A must be stored in column major format in order to have neighbouring threads access neighbouring data in texture memory.

A pseudo-code of a kernel performing step 1 of section III is shown in listing 1. First, threads load entries of the matrix B from global memory and store them in shared memory. Because the number of rows in matrix C might be higher than the number of rows in matrix B, multiple fetch cycles might be required. With the data in shared memory, the matrix-matrix multiplication is performed. Because all the threads within one element load elements in matrix B in order, those accesses are handled as broadcast and therefore typically conflict free. Conflicts can occur when the number of cells per block is not one, but the degree of the conflict is always less than the number of cells per block and therefore, its impact on the performance is limited. Finally, the data is stored. Note that as mentioned earlier, the global memory fetches by a warp are not always aligned to 128 bytes boundary but are nevertheless sequential.

```
const int ic_loc = tid/n_ed_fgpts_per_cell;
const int ifp = tid-ic_loc*n_ed_fgpts_per_cell;
const int ic = blockIdx.x*cells_per_block+ ic_loc;
const int stride_shared = cells_per_block*n_qpts;

if (tid < cells_per_block*n_ed_fgpts_per_cell && ic < n_cells)
{
        // Fetching data to shared memory
        int n_fetch_loops = (n_qpts-1)/(n_ed_fgpts_per_cell)+1;

        #pragma unroll
        for (int i=0;i<n_fetch_loops;i++)
        {
                i_qpt= i*n_ed_fgpts_per_cell+ifp;
                if (i_qpt<n_qpts)
                {
                        // Fetch the four field values of solution point i_qpt
                        m  = ic_loc *n_qpts+i_qpt;
                        m1 = ic     *n_qpts+i_qpt;

                        s_q_qpts[m] = g_q_qpts[m1]; m += stride_shared; m1 += stride_qpt;
```

```
                        s_q_qpts[m] = g_q_qpts[m1]; m += stride_shared; m1 += stride_qpt;
                        s_q_qpts[m] = g_q_qpts[m1]; m += stride_shared; m1 += stride_qpt;
                        s_q_qpts[m] = g_q_qpts[m1]; m += stride_shared; m1 += stride_qpt;
                        s_q_qpts[m] = g_q_qpts[m1];
                }
        }
}
__syncthreads();

if (tid < cells_per_block*n_ed_fgpts_per_cell && ic < n_cells)
{
        // With data in shared memory, perform matrix multiplication
        // 1 thread per flux point
        for (int i=0;i<n_qpts;i++)
        {
                m = i*n_ed_fgpts_per_cell+ifp;
                m1 = n_qpts*ic_loc+i;

                mat_entry = fetch_double(t_interp_mat_0,m);
                q0 += mat_entry*s_q_qpts[m1];   m1 += stride_shared;
                q1 += mat_entry*s_q_qpts[m1];   m1 += stride_shared;
                q2 += mat_entry*s_q_qpts[m1];   m1 += stride_shared;
                q3 += mat_entry*s_q_qpts[m1];   m1 += stride_shared;
                q4 += mat_entry*s_q_qpts[m1];
        }

        // Store in global memory
         m = ic*n_ed_fgpts_per_cell+ifp;
        g_q_ed_fgpts[m] = q0; m += stride_ed;
        g_q_ed_fgpts[m] = q1; m += stride_ed;
        g_q_ed_fgpts[m] = q2; m += stride_ed;
        g_q_ed_fgpts[m] = q3; m += stride_ed;
        g_q_ed_fgpts[m] = q4;
 }
```

Listing 1: Dense matrix multiplication kernel to compute the approximate solution at the flux points in step 1

2. Sparse matrix-matrix multiplication

When dealing with hexahedral and prismatic elements, the operator matrix A is sparse since those elements use a tensor product formulation. For hexahedral elements, the number of non-zero elements per row is constant for all the matrices M_1 to M_6. For prismatic elements, the number of non-zero elements per row in matrices M_1 to M_6, although not constant, does not vary considerably. To take advantage of this matrix property, the ELLPACK[26] format was used to store the operator matrices. For an M by N matrix with a maximum of K nonzeros per row, the ELLPACK format stores the nonzero values in a dense M by K array *Data*, where rows with fewer than K non-zeros are zero-padded. Similarly, the corresponding column indices are stored in a dense array *Indices*, again with zero, or some other flag value used for padding. Figure (12) illustrates the ELLPACK representation of a matrix with a maximum of four nonzeros per row. Again, to ensure that threads access the data and index matrices sequentially, they are stored in column major format as shown in Figure (12). A snippet of the sparse matrix-matrix multiplication kernel for step 1 is shown in listing 2. The data and index matrices are bound to texture memory. In the actual implementation, padding was added in shared memory to reduce the number of bank conflicts, which are more significant for the hexahedral element.

$$A = \begin{bmatrix} 1 & 3 & 0 & 0 & 5 & 0 \\ 7 & 0 & 9 & 5 & 2 & 0 \\ 0 & 8 & 9 & 6 & 0 & 7 \end{bmatrix} \quad Data = \begin{bmatrix} 1 & 3 & 5 & * \\ 7 & 9 & 5 & 2 \\ 8 & 9 & 6 & 7 \end{bmatrix} \quad Indices = \begin{bmatrix} 0 & 1 & 4 & * \\ 0 & 2 & 3 & 4 \\ 1 & 2 & 3 & 5 \end{bmatrix}$$

$$Data = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 1 & 7 & 8 & 3 & 9 & 9 & 5 & 5 & 6 & * & 2 & 7 \\ \hline \end{array}$$

$$Indices = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 1 & 1 & 2 & 2 & 4 & 3 & 3 & * & 4 & 5 \\ \hline \end{array}$$

Figure 12: Arrays *Data* and *Indices* which are used to represent matrix A when using the ELLPACK format

```
for (int i=0;i<n_nz0;i++)
{
        m = i*n_ed_fgpts_per_cell+ifp;
        m1 = n_qpts*ic_loc + tex1Dfetch(t_indices_0,m);

        mat_entry = fetch_double(t_data_0,m);
         q0 += mat_entry*s_q_qpts[m1];   m1 += stride_shared;
         q1 += mat_entry*s_q_qpts[m1];   m1 += stride_shared;
         q2 += mat_entry*s_q_qpts[m1];   m1 += stride_shared;
         q3 += mat_entry*s_q_qpts[m1];   m1 += stride_shared;
         q4 += mat_entry*s_q_qpts[m1];
}
```

Listing 2: Snippet of the sparse matrix multiplication kernel to compute the approximate solution at the flux points in step 1

B. Point-local operations

The point-local operations in the algorithm consist of step 5 in which the total flux is computed at each solution point in the mesh and step 11, in which the solution is advanced in time. For these steps, one thread is assigned per solution point and we allow the number of threads per thread block to be arbitrary. Because solution points in the same cells do not share data, there is not incentive to assign a number of cells per thread block. By making the thread block size a multiple of 32, the global memory loads and writes are fully coalesced thanks to the padding shown in figure (11). In order to reduce the number of registers used, step 5 was split into two kernels: one that computes the inviscid flux and one that computes the viscous flux. Although this leads to additional reads and writes from global memory, the added fetch cost is easily amortized by the increased occupancy achieved due to the lower register pressure for the inviscid kernel. Furthermore, splitting the operation allows increased overlap between communication and computation when using multiple GPUs.

C. Interface Operations

Because of the data storage pattern, which was chosen to benefit the most computationally expensive parts of the algorithm (the cell-local and point-local operations), global memory accesses tend to be uncoalesced for the interface operations. Steps 3 and 8 are separated in two kernels, one which computes the average solution and the common inviscid interface flux, and the second which computes the common viscous interface flux.

The main objective for these steps was two-fold: first, minimize the number of global memory fetches and second, minimize thread divergence. To minimize the number of global memory fetches, one thread is assigned for each flux point pair. Hence, one thread is responsible for loading the 5 fields from each flux point on each side of the interface and compute a common interface flux. Again, we do not assign a number of faces or cells per thread block and are free to change the size of the thread blocks as desired. Luckily, because neighbouring threads typically fetch data from the same cells, the L1 and L2 caches introduced in the Fermi architecture helps to reduce data fetches from DRAM. This strategy was used instead of the redundant computation strategy used by Klockner et al.[17] which would lead to additional global memory fetches. The trade-offs between the two approaches has not been thoroughly investigated.

The interface operation kernels were also split into boundary and interior face kernels in order to minimize thread divergence, which can have a significant impact on performance. Thread divergence occurs when

threads in the same warp execute different instructions. Since all the threads assigned to interior flux point pairs are executing the same instructions, there is no penalty due to thread divergence. For the boundary kernel, flux points pairs were ordered by boundary type, and threads were sequentially assigned to flux point pairs. This strategy ensures that a minimum number of warps will be divergent.

VI. Multi-GPU Implementation

The multi-GPU algorithm uses a mixed MPI-Cuda implementation that can make use of an arbitrary number of GPUs running simultaneously. The cells in the mesh are divided in a number of partitions and one GPU is assigned to each partition. The algorithm is designed to work at large scales, so the mesh is loaded in parallel and is partitioned in parallel using the graph partitioning software ParMETIS.[27] The MPICH-2 implementation of the MPI standard is used.

Because GPUs cannot communicate directly with each other, the exchange of data must be done by the CPUs controlling the GPUs. When using the FR approach, data must be exchanged at two occasions in each iteration: first, before computing the common interface solution and common interface inviscid flux values and second, before computing the common interface viscous flux values. In order to achieve good scaling with large number of GPUs, this exchange process must be done optimally.

For completeness, the three implementation approaches proposed by Jacobsen et al.[28] were considered. The performance of the three approaches will be discussed in section VII. The first approach consists of using non-blocking MPI calls without overlap between communication and computations. A pseudo-code that uses this methodology for the first data exchange is shown in listing 3.

```
// Step 1: Compute solution at flux points
calc_q_ed_fgpts_kernel<<<grid_size_1 , block_size_1 >>>(...);

// Step 2: Compute discontinuous gradient
calc_dis_grad_qpts<<<grid_size_2 , block_size_2 >>>(...);

// Step 5 (inviscid part): Compute the inviscid discontinuous flux at solution points
calc_inv_dis_flux_qpts<<<grid_size_5 , block_size_5 >>>(...);

// Pack mpi_buffer
pack_out_buffer<<<grid_size_buf , block_size_buf>>>(d_out_buffer , ... );

// Copy buffer from GPU to CPU
cudaMemcpy( h_out_buffer , d_out_buffer , size_buffer , cudaMemcpyDeviceToHost );

// Exchange data between CPUs
MPI_Isend( h_out_buffer , ... );
MPI_Irecv( h_in_buffer , ... );

// Copy buffer from CPU to GPU
cudaMemcpy( d_in_buffer , h_in_buffer , size_buffer , cudaMemcpyHostToDevice );

// Step 3...
```

Listing 3: First approach used to perform the data exchange between GPUs. Non-blocking sends and receives are used without any overlap between computation and communication

The second approach involves overlapping MPI communication with GPU computations. This can be done because step 2 and the computation of the inviscid flux in step 5 do not depend on the values being exchanged between GPUs. Hence, while the CPUs are exchanging data, the GPUs can do useful work. A schematic representation of the second strategy is shown in figure (13). A pseudo-code that uses this approach is shown in listing 4.

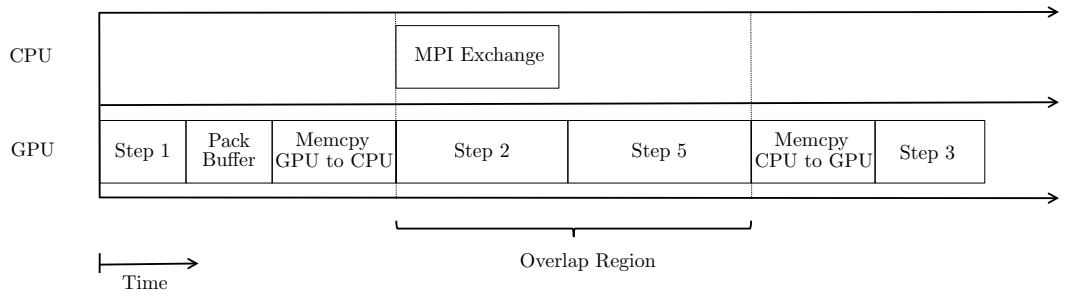

Figure 13: Schematic representation of the second approach used to perform the data exchange between GPUs. This approach overlaps MPI communications and GPU computations

```
// Step 1: Compute solution at flux points
calc_q_ed_fgpts_kernel<<<grid_size_1 , block_size_1 >>>(...);

// Pack mpi_buffer
pack_out_buffer<<<grid_size_buf , block_size_buf>>>(d_out_buffer ,...);

// Copy buffer from GPU to CPU
cudaMemcpy(h_out_buffer , d_out_buffer , size_buffer , cudaMemcpyDeviceToHost);

// Exchange data between CPUs
MPI_Isend(h_out_buffer ,...);

// Step 2: Compute discontinuous gradient
calc_dis_grad_qpts<<<grid_size_2 , block_size_2 >>>(...);

// Step 5 (inviscid part): Compute the inviscid discontinuous flux at solution points
calc_inv_dis_flux_qpts<<<grid_size_5 , block_size_5 >>>(...);

MPI_Irecv(h_in_buffer ,...);

// Copy buffer from CPU to GPU
cudaMemcpy(d_in_buffer , h_in_buffer , size_buffer , cudaMemcpyHostToDevice);

// Step 3...
```

Listing 4: Second approach used to perform the data exchange between GPUs. Non-blocking sends and receives are used and communication between CPUs is overlapped with computations performed on the GPU

The third approach uses asynchronous memory copies between the CPU and the GPU to perform steps 2 and 5 while the data is copied between the CPU and GPU. This strategy allows to overlap GPU computations, CPU communications and CPU-GPU transfers. A schematic representation of the overlapping strategy is shown in figure (14) and a pseudo-code that uses this approach is shown in listing 5.

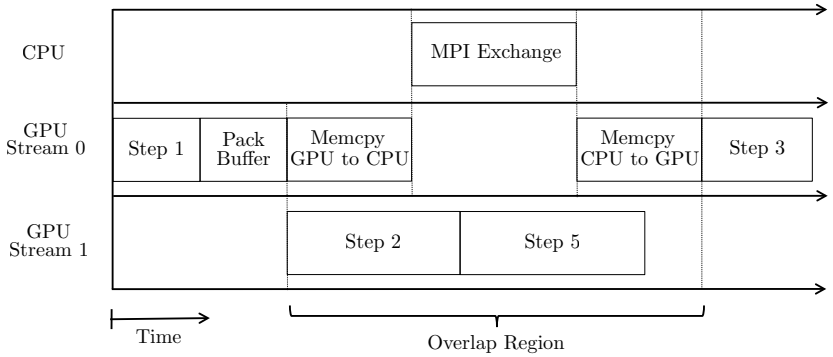

Figure 14: Schematic representation of the third approach used to perform the data exchange between GPUs. This approach overlaps MPI communications, GPU computations and CPU-GPU memory copies

```
// Step 1: Compute solution at flux points
calc_q_ed_fgpts_kernel<<<grid_size_1 , block_size_1 >>>(...);

// Pack mpi_buffer
pack_out_buffer<<<grid_size_buf , block_size_buf>>>(d_out_buffer , ...);

// Copy buffer from GPU to CPU
cudaMemcpyAsync(h_out_buffer , d_out_buffer , size_buffer , cudaMemcpyDeviceToHost , STREAM[0]);

// Step 2: Compute discontinuous gradient
calc_dis_grad_qpts<<<grid_size_2 , block_size_2 ,0 ,STREAM[1]>>>( );

// Step 5 (inviscid part): Compute the inviscid discontinuous flux at solution points
calc_inv_dis_flux_qpts<<<grid_size_5 , block_size_5 ,0 ,STREAM[1]>>>( );

// Exchange data between CPUs
MPI_Isend(h_out_buffer , );

MPI_Irecv(h_in_buffer , );

// Copy buffer from CPU to GPU
cudaMemcpyAsync(d_in_buffer , h_in_buffer , size_buffer , cudaMemcpyHostToDevice ,STREAM[0]);

// Step 3...
```

Listing 5: Third approach used to perform the data exchange between GPUs. This approach overlaps MPI communications, GPU computations and CPU-GPU memory copies

VII. Performance Analysis

A. Single GPU

In order to compare the GPU and CPU implementations of our solver, the solution for the viscous flow over a sphere at a Reynolds number of 100 and Mach 0.2 was calculated. The solution was advanced in time using a low-storage 4th order Runga-Kutta scheme.[23] The performance of the CPU and GPU codes are compared by measuring the time taken to run 100 time advancement iterations. Three different grid types were used, each one containing one element type, namely tetrahedra, hexahedra and prisms. The number of cells in each grid, along with the total number of degrees of freedom (DOFs) for the different orders of accuracy considered is shown in table (1). All GPU results were obtained using a Tesla C2050 GPU, using double precision with ECC turned on. The CPU results were obtained on a single core of a Xeon x5670 2.93 GHz processor. In order to ensure a fair comparison, every effort was made to maximize the performance of the CPU version of the code, which uses the Intel Math Kernel Library version 10.3 to perform the dense matrix operations and the Optimized Sparse Kernel Interface (OSKI)[29] for the sparse matrix operations.

Overall performance of the GPU code is shown in figure (15) for the different element types and orders of accuracy. For tetrahedral elements, the overall performance of the GPU implementation reaches 116 Gflops for a 6th order accurate solution. Also, for tetrahedra, as the order of accuracy is increased, performance of the GPU code increases gradually. This can be explained by the fact that as the order of accuracy increases, there is an increase in the amount of work per degree of freedom and also, more data reuse. The overall performance of the GPU code on hexahedral and prismatic element is significantly less than for tetrahedral elements. This can be explained by the fact that the operator matrices for prisms and hexahedra are very sparse, hence the element-local operations tend to have a much lower instruction to memory access ratio.

Figure (16) shows the overall speedup of the GPU code compared to a serial version of the CPU code running on a single core of the Xeon x5670 processor. The GPU code achieves speedups of at least 25 for all orders of accuracy tested and for all element types. Thus, even if the GPU code running on hexahedra and prisms does not reach the level of performance attained when running on tetrahedra, the speedup is still significant for all element types. The best speedup is observed for a 4th order accurate simulation with hexahedral elements. This can be explained by the fact that for that order, the number of solution points and flux points per cell are 64 and 96 respectively, both of which are multiples of 32, the warp size. Hence, for that case, all the global memory reads and writes in the cell-local kernels are performed in a fully coalesced manner.

Element Type	Order of Accuracy	# of cells	# of DOFs
Tet	3	27,915	279,150
Tet	4	27,915	558,300
Tet	5	27,915	977,025
Tet	6	27,915	1,563,240
Hex	3	20,736	559,872
Hex	4	20,736	1,327,104
Hex	5	20,736	2,592,000
Hex	6	10,206	2,204,496
Prism	3	40,500	729,000
Prism	4	40,500	1,620,000
Prism	5	20,412	1,530,900
Prism	6	20,412	2,571,912

Table 1: Number of cells used to measure the performance of the GPU and CPU codes for the various element types and orders of accuracy

The performance of the double precision high-order Navier-Stokes developed in this work compares favourably to the performance of double precision Navier-Stokes solvers developed by other groups. Unfortunately, we could not find another GPU-enabled unstructured high-order Navier-Stokes solver hence, comparison will be made with low-order unstructured Navier-Stokes solvers. Corrigan et al.[30] showed an unstructured grid solver achieving a speedup of 7.4x over a serial computation for double precision, comparing a NVIDIA Tesla 10 card to a Core 2 processor . Kampolis et al.[31] achieved a 19.5x speedup for their Navier-Stokes solver on a GTX285 compared to a Code 2 Duo 3.8GHz. DeVito et al.[32] also obtained a 19.5x speedup (comparing a Tesla C2050 with a single core of a Xeon X5650 processor) using Liszt, a domain specific language for building mesh-based PDE solvers. The performance of the GPU code developed in this work outperforms the other solvers. This can be explained by the fact that high-order solvers typically have more operations per DOFs than their low-order counterparts, hence benefit the most from the high computational throughput of GPUs.

Figures (17) through (19) shows the performance (in terms of Gflops) of the kernels responsible for the cell-local, point-local and interface operations. For tetrahedral elements, the cell-local operations achieve an impressive level of performance (up to 138 Gflops). For hexahedral and prismatic elements, those operations cannot reach this level of performance because of the sparsity of the operator matrices.

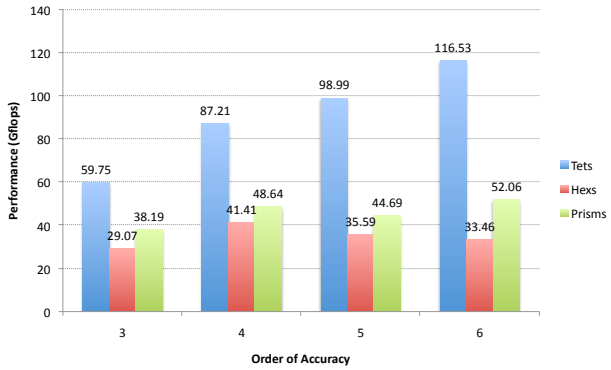

Figure 15: Performance in Gflops of the single-GPU algorithm. Computations were performed using double precision.

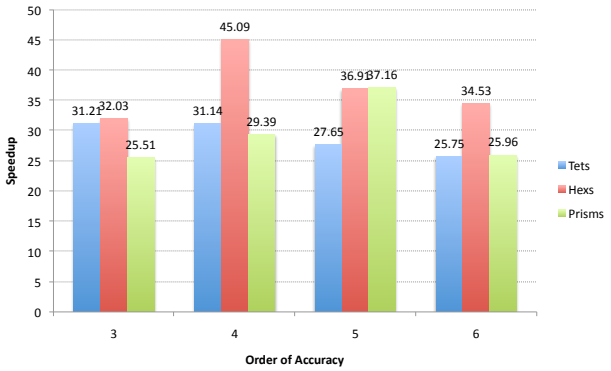

Figure 16: Speedup of the single-GPU algorithm relative to a serial computation on a single core of the Xeon x5670 CPU

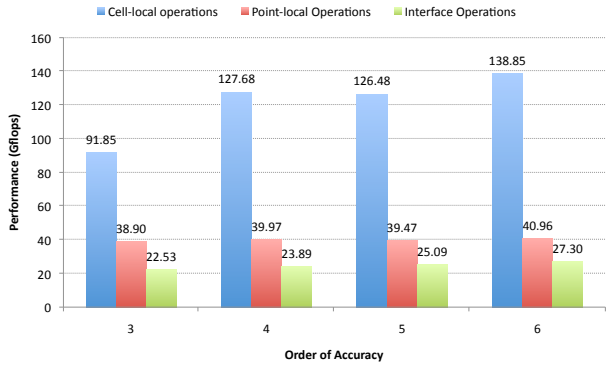

Figure 17: Performance in Gflops of each part of the FR algorithm, when running on tetrahedral elements

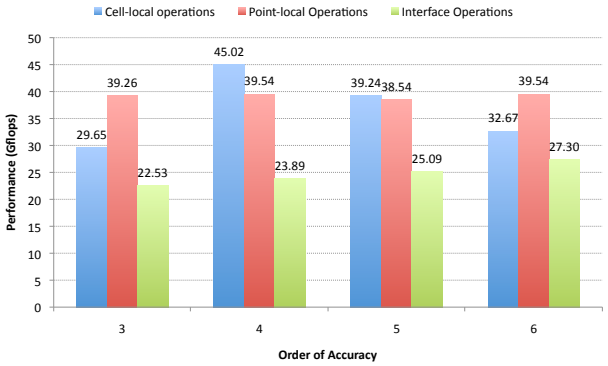

Figure 18: Performance in Gflops of each part of the FR algorithm, when running on hexahedral elements

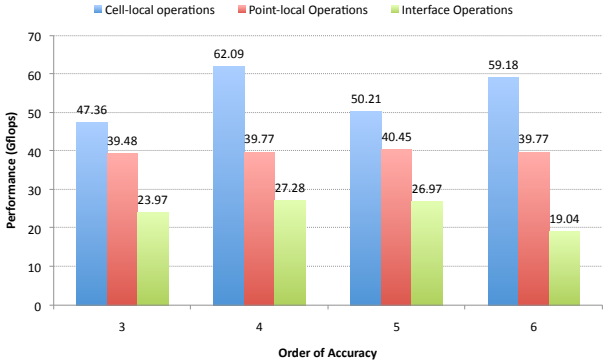

Figure 19: Performance in Gflops of each part of the FR algorithm, when running on prismatic elements

B. Multi GPU

The multi-GPU version of the code was tested on a 16-node cluster at NVIDIA, with each node having two Tesla C2070 GPUs and two Intel Xeon x5670 processors. The nodes are SuperMicro 6061 compute nodes and each GPU card is connected via a PCIE Gen 2 x16 connection, which leaves a x4 connection for the single port QDR Infiniband interface. To illustrate the effect of overlapping GPU computations with MPI communication and GPU transfers, the scalability of the code on up to 32 GPUs was investigated using the three strategies discussed in section VI. A sixth order accurate solution was obtained on a mesh with 55,947 tetrahedral elements (3.13 million DOFs). The speedup versus the number of GPUs is shown in figure (20). The results indicate that the use of CUDA streams to overlap GPU computations, GPU transfers and MPI communication has a beneficial impact on the scaling. In fact, when running on 32 GPUs, the fully overlapped version achieves a speedup of 24.7 while the version without any overlap achieves a speedup of 20.6. Similar trends are observed for all orders of accuracy and element types. Based on these results, the third data exchange approach was the one used in the multi-GPU code as it leads to the best scaling. It should be noted that when running on 32 GPUs, the 6th order simulation on tetrahedral elements reaches an overall performance of 24.7×116.53 Gflops $= 2.88$ Teraflops.

To investigate the scalability of the code, a weak scalability study was conducted for all orders of accuracy and for all element types. The code was run on up to 32 GPUs by keeping the number of cells per GPU constant. The number of cells per GPU was $27,915 \pm 1\%$ for the runs on tetrahedral grids, $10,206 \pm 2\%$ for the runs on hexahedral meshes and $20,412 \pm 1\%$ for the runs on prismatic grids. Figures (21) through (23) shows the efficiency of the computation versus the number of GPUs (denoted by χ), calculated as the time taken by χ GPUs divided by the time taken by a single GPU. Results indicate very good weak scalability for all orders of accuracy and all element types, as the cluster efficiency is close to 90% for all the cases considered.

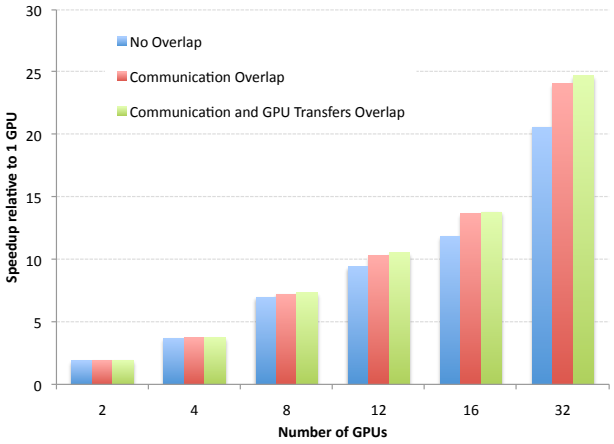

Figure 20: Speedup relative to 1 GPU versus the number of GPUs for a 6th order accurate simulation running on a mesh with 55947 tetrahedral elements. The three data exchange strategy discussed in section VI are compared

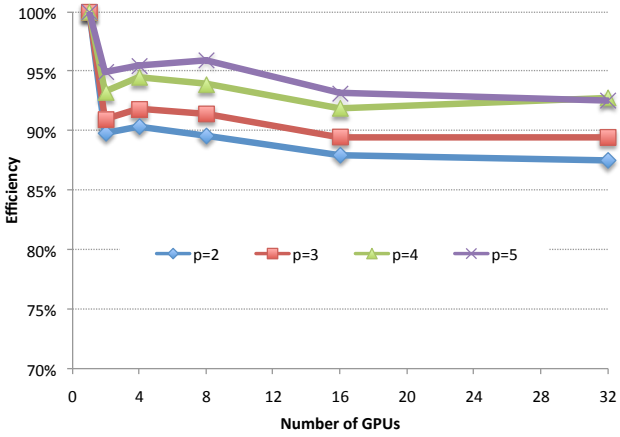

Figure 21: Weak scalability of the multi-GPU implementation running on tetrahedral elements

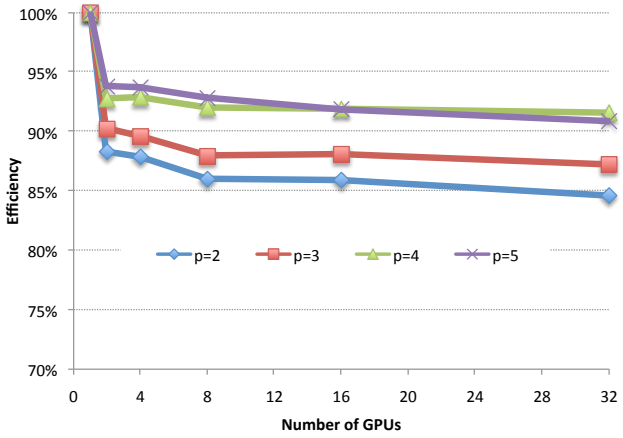

Figure 22: Weak scalability of the multi-GPU implementation running on hexahedral elements

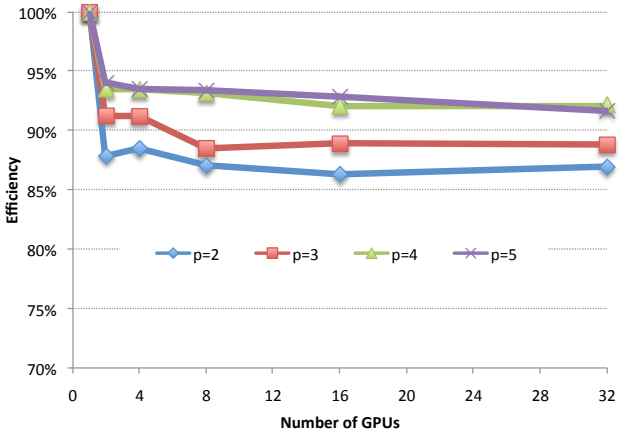

Figure 23: Weak scalability of the multi-GPU implementation running on prismatic elements

VIII. Numerical Experiments

Additional numerical experiments were conducted to demonstrate the capability of the multi-GPU algorithm. For all cases, the CPU and GPU versions of the code produced identical results. Results were obtained using the VCJH schemes defined in references 15 and 16. More specifically, the VCJH schemes used have values of c_{sd} and κ_{sd} for hexahedral and prismatic elements, and κ_{dg} for tetrahedral elements.

A. Flow over sphere at Reynolds Number 118

As a first test case, the solution for the steady flow over a sphere at a Reynolds number of 118 and a Mach number of 0.2 was obtained. The code was run on a personal desktop computer built in our lab which has 3 Tesla C2050 GPUs. The mesh used is shown in figure (24). It contains 38,500 prisms to capture the boundary layer and 99,951 tetrahedral elements in the rest of the domain. A 4th order accurate simulation was run, hence the total number of DOFs was 3.54 million. The solution was advanced in time using a low-storage 4th order Runga-Kutta scheme.[23] The contours of Mach number are shown in figure (25). For this test case, the simulation running on 3 GPUs on our desktop machine achieved the same performance as the multi-CPU version of the code (which uses MPI for communication between cores) running on 15 nodes of the NVIDIA cluster (where each node has two 6-core Xeon x5670 processors).

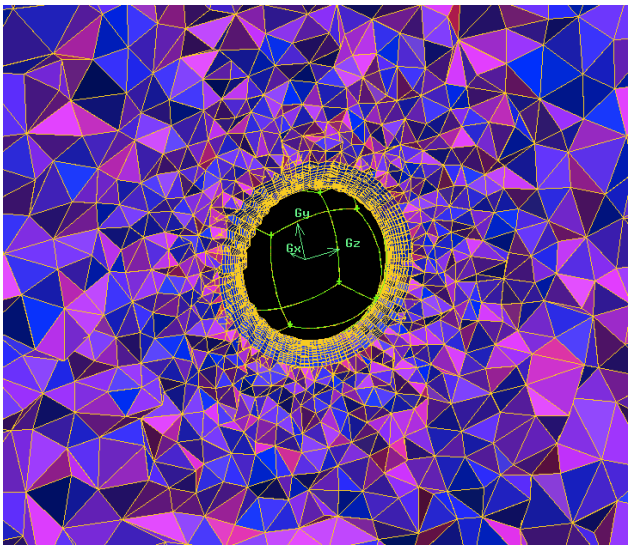

Figure 24: Mixed mesh used to simulate the viscous flow over a sphere at a Reynolds number 118 and a Mach number of 0.2

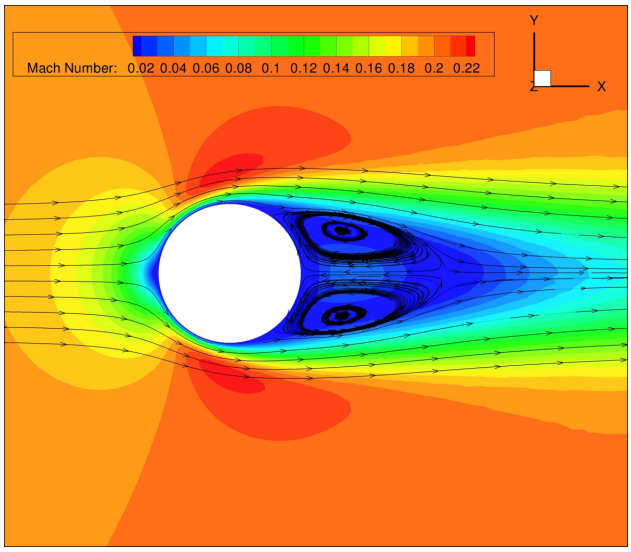

Figure 25: Contours of Mach number obtained using a 4th order accurate solution on a mixed mesh comprised of prismatic and tetrahedral elements. The Reynolds number is 118 and the Mach number 0.2

B. Transitional Flow over SD7003 airfoil at Reynolds 60000

The second test case considered was the transitional flow over the SD7003 airfoil at a Reynolds number of 60000, an angle of attack of 4 degrees and a Mach number of 0.2. For this case, a 4th order accurate solution was obtained on a hexahedral mesh containing 331,776 cells, for a total of 21.2 million DOFs. The instantaneous solution after 400,000 Runga-Kutta iterations is shown in figure (26) and was obtained in approximately 15 hours using the multi-GPU code running on 16 Tesla C2070 GPUs. The multi-CPU version of the code took 102 hours when running on 32 Xeon x5670 processors.

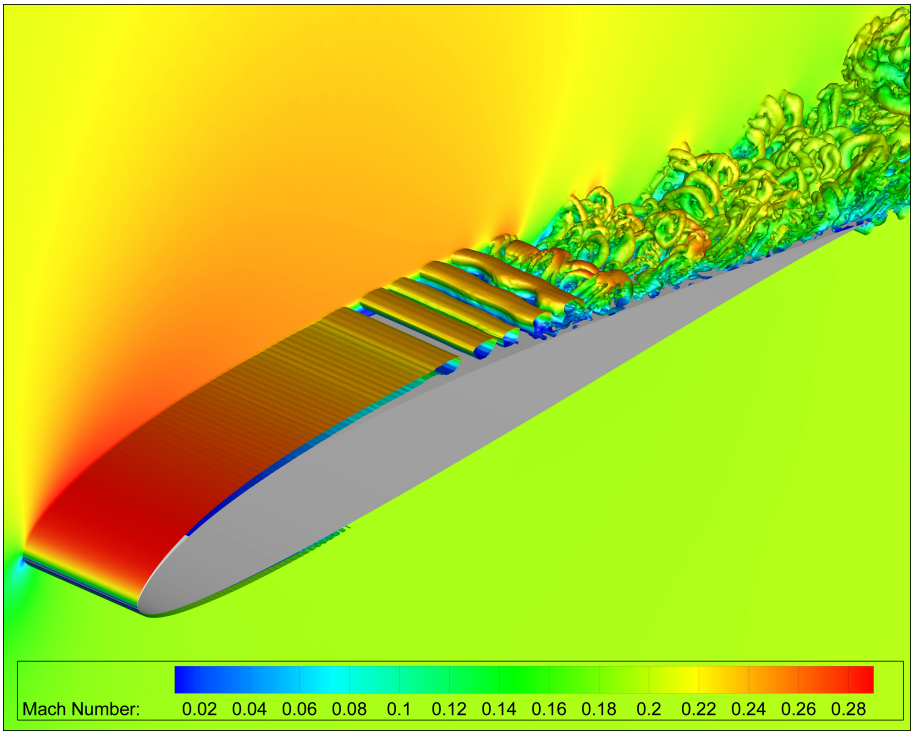

Figure 26: Instantaneous iso-surfaces of Q-criterion colored by Mach number. Flow over SD7003 airfoil at a Reynolds number of 60000, an angle of attack of 4 degrees and a Mach number of 0.2

IX. Conclusions

This work discusses the development of a three-dimensional, high-order, compressible viscous flow solver for mixed unstructured grids that can run on multiple GPUs. The study demonstrates that high-order, unstructured Navier-Stokes solvers can achieve significant performance improvements from the use of GPUs. The single GPU algorithm developed in this work achieves speed-ups of at least 25 relative to a serial computation on a current generation CPU for all orders of accuracy and element types investigated. The multi-GPU solver scales well and, when running on 32 GPUs, achieves a sustained performance of 2.8 Teraflops (double precision) for 6th order accurate simulations on tetrahedral elements. The efficient use of texture memory and shared memory, the ability to vary the number of cells per thread block for the cell-local operations and the data layout which leads to efficient global memory accesses allow the code to reach such levels of performance. Although the GPU implementation is not optimal in several ways, it is able to make use of the high computational power of GPUs to significantly accelerate a complex high-order Navier-Stokes solver for mixed unstructured grids. As demonstrated in section VIII, a small to medium size GPU cluster can attain the same level of performance as a large and costly to maintain CPU cluster. The authors believe that within the foreseeable future, these developments will enable high-order accurate large eddy simulations (LES) for realistic industrial applications.

Acknowledgments

The authors would like to acknowledge the support for this work provided by the Stanford Graduate Fellowships program, the Natural Science and Engineering Research Council (NSERC) of Canada, the Fonds Québécois de la Recherche sur la Nature et les Technologies (FQRNT), the National Science Foundation (grants 0708071 and 0915006)), the Air Force Office of Scientific Research (grants FA9550-07-1-0195 and FA9550-10-1-0418) and the National Science Foundation Graduate Research Fellowship Program for supporting this work. The authors would also like to thank Thomas Reed and Stan Posey from NVIDIA for their technical support and hardware donations.

References

[1]Reed, W. and Hill, T., "Triangular mesh methods for the neutron transport equation," *Los Alamos Report LA-UR-73-479*, 1973.

[2]Cockburn, B. and Shu, C., "for Conservation Laws II: General Framework," *Mathematics of Computation*, Vol. 52, No. 186, 1989, pp. 411–435.

[3]Cockburn, B., Lin, S., and Shu, C., "TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws III: one-dimensional systems," *Journal of Computational Physics*, Vol. 84, No. 1, 1989, pp. 90–113.

[4]Cockburn, B., Hou, S., and Shu, C., "The Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws IV: the multidimensional case," *Math. Comp*, Vol. 54, No. 190, 1990, pp. 545–581.

[5]Cockburn, B. and Shu, C., "The Runge-Kutta Discontinuous Galerkin Method for Conservation Laws V* 1:: Multidimensional Systems," *Journal of Computational Physics*, Vol. 141, No. 2, 1998, pp. 199–224.

[6]Hesthaven, J. and Warburton, T., *Nodal discontinuous Galerkin methods: algorithms, analysis, and applications*, Springer Verlag, 2007.

[7]Kopriva, D. A. and Kolias, J. H., "A Conservative Staggered-Grid Chebyshev Multidomain Method for Compressible Flows," *Journal of Computational Physics*, Vol. 125, 1996, pp. 244–261.

[8]Liu, Y., Vinokur, M., and Wang, Z. J., "Spectral difference method for unstructured grids I: Basic formulation," *Journal of Computational Physics*, Vol. 216, 2006, pp. 780–801.

[9]Wang, Z., Liu, Y., May, G., and Jameson, A., "Spectral Difference Method for Unstructured Grids II: Extension to the Euler Equations," *Journal of Scientific Computing*, Vol. 32, 2007, pp. 45–71.

[10]Liang, C., Kannan, R., and Wang, Z. J., "A p-Multigrid Spectral Difference Method with explicit and implicit smoothers on unstructured triangular grids," *Computers and Fluids*, Vol. 38, 2009, pp. 254–265.

[11]Castonguay, P., Liang, C., and Jameson, A., "Simulation of Transitional Flow over Airfoils using the Spectral Difference Method," *AIAA P.*, Vol. 2010-4626, 2010.

[12]Huynh, H., "A Flux Reconstruction Approach to High-Order Schemes Including Discontinuous Galerkin Methods," *AIAA P.*, Vol. 4079, 2007, 18th AIAA Computational Fluid Dynamics Conference, Miami, FL, Jun 25–28, 2007.

[13]Huynh, H., "A Reconstruction Approach to High-Order Schemes Including Discontinuous Galerkin for Diffusion," *AIAA P.*, Vol. 403, 2009, 47th AIAA Aerospace Sciences Meeting, Orlando, FL, Jan 5–8, 2009.

[14]Jameson, A., "A Proof of the Stability of the Spectral Difference Method for All Orders of Accuracy," *J. Sci. Comput.*, Vol. 45, October 2010, pp. 348–358.

[15]Vincent, P. E., Castonguay, P., and Jameson, A., "A New Class of High-Order Energy Stable Flux Reconstruction Schemes," *Journal of Scientific Computing*, Vol. 47, 2011, pp. 50–72.

[16]Castonguay, P., Vincent, P. E., and Jameson, A., "A New Class of High-Order Energy Stable Flux Reconstruction Schemes for Conservation Laws on Triangular Grids," *Journal of Scientific Computing*, 2011, DOI: 10.1007/s10915-011-9505-3.

[17]Klockner, A., Warburton, T., Bridge, J., and Hesthaven, J., "Nodal discontinuous Galerkin methods on graphics processors," *Journal of Computational Physics*, Vol. 228, No. 21, 2009, pp. 7863–7882.

[18]Jameson, A., Vincent, P. E., and Castonguay, P., "On the Non-Linear Stability of Flux Reconstruction Schemes," *Journal of Scientific Computing*, 2011, DOI: 10.1007/s10915-011-9490-6.

[19]Castonguay, P., Vincent, P. E., and Jameson, A., "Application of Energy Stable Flux Reconstruction Schemes to the Euler Equations," *AIAA P.*, Vol. 2011-0686, 2011.

[20]Bassi, F. and Rebay, S., "A High-Order Accurate Discontinuous Finite Element Method for the Numerical Solution of the Compressible Navier-Stokes Equations* 1," *Journal of Computational Physics*, Vol. 131, No. 2, 1997, pp. 267–279.

[21]Roe, P. L., "Approximate Riemann Solvers, Parameter Vectors and Difference Schemes," *Journal of Computational Physics*, Vol. 43, 1981, pp. 357–372.

[22]Raviart, P. and Thomas, J., "A mixed hybrid finite element method for the second order elliptic problems," *Mathematical Aspects of the Finite Element Method, Lectures Notes in Mathematics*, 1977.

[23]Carpenter, M. H. and Kennedy, C., "Fourth-order 2N-storage Runge-Kutta schemes," Tech. Rep. TM 109112, NASA, NASA Langley Research Center, 1994.

[24]NVIDIA Corporation, "CUDA Programming Guide," Version 3.2, 2010.

[25]NVIDIA Corporation, "CUDA CUBLAS Library," version 3.2, 2010.

[26]Grimes, R., Kincaid, D., and Young, D., "ITPACK 2.0 User's Guide, CNA-150," *Center for Numerical Analysis, University of Texas, Austin, Texas*, Vol. 78712, 1979.

[27]Karypis, G., Schloegel, K., and Kumar, V., "PARMETIS 2.0: Parallel graph partitioning and sparse matrix ordering library," Tech. rep., Technical report, Department of Computer Science, University of Minnesota, 1998.

[28]Jacobsen, D., Thibault, J., and Senocak, I., "An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters," *Mechanical and Biomedical Engineering Faculty Publications and Presentations*, 2010, pp. 5.

[29]Vuduc, R., Demmel, J., and Yelick, K., "OSKI: A library of automatically tuned sparse matrix kernels," *Journal of Physics: Conference Series*, Vol. 16, IOP Publishing, 2005, p. 521.

[30]Corignan, A., Camelli, F., Lohner, R., and Wallin, J., "Running Unstructured Grid CFD Solvers on Modern Graphics Hardware," *AIAA P.*, Vol. 2009-4001, June 2009, 19th AIAA Computational Fluid Dynamics.

[31]Kampolis, I., Trompoukis, X., Asouti, V., and Giannakoglou, K., "CFD-based analysis and two-level aerodynamic optimization on graphics processing units," *Computer Methods in Applied Mechanics and Engineering*, Vol. 199, No. 9-12, 2010, pp. 712–722.

[32]Devito, Z., Joubert, N., Palacios, F., Oakley, S., Medina, M., Barrientos, M., Elsen, E., Ham, F., Aiken, A., Duraisamy, K., Darve, E., Alonso, J., and Hanrahan, P., "Liszt: A Domain Specific Language for Building Portable Mesh-based PDE Solvers," *In Proceedings of the Conference on Supercomputing*, November 2011.